Pelican User Guide

1.0

Generated by Doxygen 1.8.2

Tue Jan 22 2013 11:07:32

Contents

1 Introduction **1**

1.1 Overview . . . 1

1.1.1 Structure of this Guide . . . 1

1.2 The Pelican Framework . . . 1

1.3 Using the Pelican API . . . 1

1.4 Pelican Use Cases . . . 2

2 The Pelican Framework **3**

2.1 Introduction . . . 3

2.1.1 The Modules Required for a Pelican Application . . . 4

2.2 Framework Modes . . . 4

3 Getting Started (Tutorial) **5**

3.1 Linking against Pelican . . . 6

3.1.1 Example . . . 6

3.1.2 Pelican CMake Module . . . 6

3.2 Importing Data I: Concepts, Data Emulators and Chunkers . . . 7

3.2.1 Introduction . . . 7

3.2.2 Input Data Types . . . 7

3.2.2.1 Stream Data . . . 7

3.2.2.2 Service Data . . . 7

3.2.3 An Example Input Stream . . . 7

3.2.3.1 Emulating Input Stream Data . . . 8

3.2.4 Chunkers . . . 10

3.2.4.1 Tutorial Chunker . . . 10

3.3 Importing Data II: The Server, Client and Data Adapters . . . 13

3.3.1 Introduction . . . 13

3.3.2 The Pelican Server . . . 13

3.3.2.1 Attaching chunkers to the server . . . 13

3.3.3 Data Clients . . . 13

3.3.3.1 Specifying the Data Client . . . 13

3.3.3.2 Installing Chunkers into the DirectStreamDataClient . . . 14

3.3.3.3 Installing an Adapter into a DataClient . . . 14
3.3.4 Adapters . . . 15
3.3.4.1 Data-Blobs . . . 15
3.3.4.2 Tutorial Data-Blob . . . 15
3.3.4.3 Tutorial Adapter . . . 16
3.4 Processing Data: Pipelines and Modules . . . 18
3.4.1 Introduction . . . 18
3.4.2 Overview . . . 18
3.4.3 Pipeline Modules . . . 18
3.4.3.1 Tutorial Module . . . 18
3.4.4 The Pipeline . . . 20
3.4.4.1 Tutorial Pipeline . . . 20
3.5 Data Output . . . 22
3.5.1 Introduction . . . 22
3.5.2 Data Output from a Pipeline . . . 22
3.5.3 Data-Blob Output Requirements . . . 22
3.5.3.1 Tutorial Data Blob Output . . . 22
3.5.4 The Output Stream Manager . . . 23
3.5.4.1 The DataBlobFile . . . 23
3.5.4.2 The PelicanTCPBlobServer . . . 23
3.5.4.3 Tutorial Output . . . 23
3.5.5 Reading from the PelicanTCPBlobServer . . . 24
3.6 Tutorial Conclusion . . . 25
3.6.1 The Emulator Binary . . . 25
3.6.1.1 Running the Emulator . . . 25
3.6.2 The Server Binary . . . 25
3.6.2.1 Server XML . . . 26
3.6.2.2 Running the Server . . . 26
3.6.3 The Pipeline Binary . . . 26
3.6.3.1 Pipeline XML . . . 27
3.6.3.2 Running the Pipeline . . . 28
3.6.4 Data Output . . . 28

4 Pelican Reference **31**

4.1 Data Chunkers . . . 32
4.1.1 Introduction . . . 32
4.1.2 Overview . . . 32
4.1.3 Configuration . . . 32
4.1.4 Built In Chunkers . . . 32
4.1.4.1 FileChunker . . . 32

4.1.5 Example . . . 33
4.1.6 Testing your Chunker . . . 34
4.2 Data Clients . . . 35
4.2.1 Introduction . . . 35
4.2.2 Overview . . . 35
4.2.3 Configuration . . . 35
4.2.4 Specialised data clients . . . 35
4.2.4.1 The PelicanServerClient class . . . 35
4.2.4.2 The FileDataClient class . . . 35
4.2.4.3 The DirectStreamDataClient class . . . 36
4.3 Data Adapters . . . 36
4.3.1 Introduction . . . 36
4.3.2 Overview . . . 36
4.3.3 Configuration . . . 37
4.3.4 Example . . . 37
4.4 Data Blobs . . . 39
4.4.1 Introduction . . . 39
4.4.2 Overview . . . 39
4.4.3 Example . . . 39
4.5 Pipeline Modules . . . 41
4.5.1 Introduction . . . 41
4.5.2 Overview . . . 41
4.5.3 Configuration . . . 41
4.5.4 Example . . . 41
4.6 Pipelines . . . 44
4.6.1 Introduction . . . 44
4.6.2 Overview . . . 44
4.6.2.1 The init() method . . . 44
4.6.2.2 The run() method . . . 44
4.6.3 Example . . . 45
4.7 Output Streamers . . . 47
4.7.1 Overview . . . 47
4.7.1.1 The Pelican DataBlobFile format . . . 47
4.7.1.2 The TCP Blob Server (PelicanTCPBlobServer) . . . 47
4.7.2 Custom OutputStreamers . . . 47
4.7.2.1 . . . 48
4.8 Configuration Files . . . 49
4.8.1 Introduction . . . 49
4.8.2 File Structure . . . 49
4.8.2.1 Example . . . 50

4.8.3 Importing Configuration Data . . . 50
4.8.3.1 Importing Nodesets . . . 50
4.8.3.2 Importing Files . . . 51
4.8.3.3 Importing Remote Files . . . 51
4.8.4 Obtaining Configuration Data . . . 51
4.8.5 Common Module Options . . . 52
4.8.6 Common Chunker Options . . . 52
4.8.7 Common Data Client Options . . . 52
4.9 Data Emulation . . . 54
4.9.1 Introduction . . . 54
4.9.2 Overview . . . 54
4.9.3 Configuration . . . 54
4.9.4 Example . . . 55
4.10 Building Pelican Binaries . . . 57
4.10.1 Introduction . . . 57
4.10.2 Pipeline Binaries . . . 57
4.10.2.1 Single Pipeline . . . 57
4.10.2.2 Multiple Pipelines . . . 57
4.10.3 Server Binary . . . 58
4.10.3.1 Notes (refer to references in the code): . . . 58
4.10.4 The direct stream client . . . 59
4.11 Building the Pelican library . . . 60
4.11.1 Build Options . . . 60
4.12 Testing Utilities . . . 61
4.12.1 The Pelican test utility library . . . 61
4.12.2 Testing with CppUnit . . . 61

Chapter 1

Introduction

1.1 Overview

Pelican is an efficient, lightweight C++ library for quasi-real time data processing. The library provides a framework to separate the acquisition and processing of data, allowing the scalability and flexibility to fit a number of scenarios.

With its origin in radio astronomy, processing data as it arrives from a telescope, Pelican was originally an acronym for the *Pipeline for Extensible, Lightweight Imaging and CAlibratioN*. However, the framework is sufficiently generic to be useful to any application that requires the efficient processing of incoming data streams.

1.1.1 Structure of this Guide

This user guide is divided into three main sections:

- A brief introduction the Pelican library framework.
- A getting started guide explaining the various concepts used in Pelican, as well as providing a tutorial walk-through of a basic application use case.
- Reference documentation describing the components of Pelican in more detail, and how they can be used.

1.2 The Pelican Framework

The Pelican framework provides a system to split one or more incoming data streams into well defined chunks. These chunks are then passed on to any number of parallel computational pipelines for processing. As a chunk is passed to a pipeline only when it is available to process the data, an efficient load balancing can be achieved.

Pelican also provides facilities to export and view data streams as processing completes. For example, processed data from a series of pipelines could be easily streamed to another Pelican server for aggregation and distribution to other pipelines for further processing.

1.3 Using the Pelican API

Applications using Pelican are written by utilising or implementing a number of components which adhere to a well defined API and describe:

- How input data streams should be split into parallelisable chunks.
- The structures and methods for de-serialising chunks of input data.

- The processing to be performed.
- The data products to deliver after processing.

Pelican applications are not, in general, intended to be highly interactive. Instead, components can be configured using XML parameter files, which are read on initialisation.

1.4 Pelican Use Cases

Applications written to use the Pelican library can be run in two different configurations, depending on the data processing requirements. Because of the modular design of the library, it is easy to switch between the two as needs arise.

For very high throughput and heavy data processing, the most appropriate way to use Pelican in its server-client mode, which provides a highly scalable processing framework. Data is divided into chunks that are buffered by the Pelican server. Pelican pipelines run on other machines, where data clients request chunks of data from the server as often as the pipelines become available. If the existing pipelines cannot keep up with the input data rate, then more pipelines can be added as required.

For more modest data processing needs, a single machine can be used by connecting a processing pipeline directly to the input data stream.

Chapter 2

The Pelican Framework

2.1 Introduction

The Pelican framework is designed to allow the flexible deployment of hardware resources to cope with the data rates and data processing requirements of an incoming data stream. Pelican maintains three distinct layers with well defined interfaces between them, namely data acquisition, data processing, and data output.

Within these layers the various functional elements (e.g. chunking of the data) are modularised, to allow easy reuse and re-deployment.

The diagram below provides an overview of the various components which make up the Pelican framework, and the two deployment options for Pelican applications. The left diagram shows the server-client configuration, where a Pelican data server collects the various input data streams and makes them available to one or more processing pipelines. The diagram on the right shows a configuration where pipelines connect directly to data streams using a direct data client.

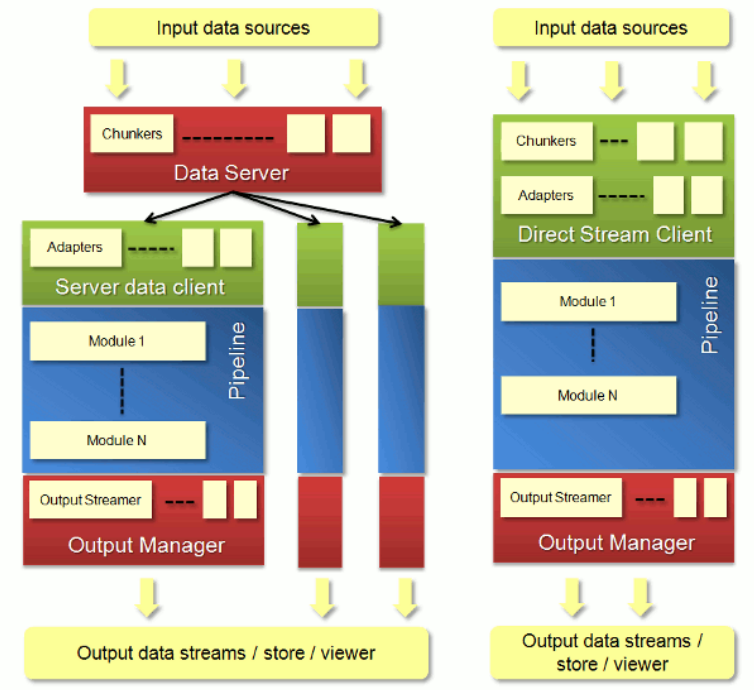

Figure 2.1: The Pelican Framework

2.1.1 The Modules Required for a Pelican Application

All of the following functional units must be present in a Pelican application:

- **Chunkers:** Describe how the input streams should be split.
- **Adapters:** Specify how chunks of stream data should be de-serialised into data objects (DataBlobs).
- **DataBlobs:** Describe the data structures used for data processing.
- **Processing Modules:** Define the computational units to be carried out.
- **Pipelines:** Describe the type and order of processing modules.
- **Output Streamers:** Define what to do with data products from pipeline processing.

Each one has a base class from which to inherit, and is the mechanism by which you extend Pelican to your needs. More details can be found in the Getting Started Guide.

2.2 Framework Modes

- **Server mode**: A Pelican server binary receives the input data streams, and pipeline binaries receive their data from the server by means of a request.
- **Direct input mode**: Pipeline binaries receive data streams directly.

Chapter 3

Getting Started (Tutorial)

This chapter is intended to introduce new users to the various components of the Pelican framework.

As described in the introduction, Pelican requires that you implement a number of C++ classes that inherit from a number of abstract interfaces. As well as explaining the relevant concepts, this chapter also describes how Pelican can be used to process a simple example data stream of UDP packets. This description takes the form of an embedded tutorial that illustrates how to implement the required components, and to build a working system.

- Linking against Pelican.
- Importing Data I: Concepts, Data Emulators and Chunkers.
- Importing Data II: The Server, Client and Data Adapters.
- Processing Data: Pipelines and Modules.
- Data Output.
- Tutorial Conclusion: Putting it all together.

3.1 Linking against Pelican

Pelican is a C++ library and can be incorporated into your build system in the same way as you would incorporate any other C++ library.

After obtaining a copy of Pelican (either from a binary distribution or building from source), you should be able to find two Pelican shared libraries (`pelican` and `pelican-testutils`) and a folder containing a set of header (.h) files which can be used to compile and link your project.

Of the two library files, the `pelican` library contains the core framework, and the `pelican-testutils` library provides a number of convenience classes that are useful when writing unit tests for the various Pelican components.

Note

Currently Pelican is targeted to run on Linux. While there are no known reasons that would prevent Pelican building on Windows or Mac OS systems, these platforms should be considered currently untested and unsupported.

3.1.1 Example

In order to link a simple application which uses Pelican:

```
g++ -o myApp myApp.cpp -I/pelican/include/path -lpelican
```

Linking a simple application using pelican and the test utility library:

```
g++ -o myUnitTest myUnitTest.cpp -I/pelican/include/path -lpelican -lpelican-
      testutils
```

3.1.2 Pelican CMake Module

If your project uses a CMake (`www.cmake.org`) build system, you can use the FindPelican cmake macro. This macro is available from the cmake folder of the Pelican source tree, the Pelican cmake module folder of an installed version of Pelican, or it can be downloaded from `https://wiki.oerc.ox.ac.uk/svn/pelican/cmake/FindPelican.cmake`. By placing this macro in the cmake module path, it can be used like any other cmake macro, and it will define the following cmake variables:

- `PELICAN_FOUND`: Set to true if Pelican is found.
- `PELICAN_CMAKE_MODULE_DIR`: Location of Pelican specific cmake modules.
- `PELICAN_INCLUDE_DIR`: Location of Pelican header files.
- `PELICAN_INCLUDES`: Set of include directories needed by Pelican (including dependencies).
- `PELICAN_LIBRARY`: The Pelican library.
- `PELICAN_TESTUTILS_LIBRARY`: The Pelican test utility library.
- `PELICAN_LIBRARIES`: The set of libraries required for linking with Pelican (includes dependencies).

3.2 Importing Data I: Concepts, Data Emulators and Chunkers

3.2.1 Introduction

This section of the guide explains how to import data into Pelican pipelines for subsequent processing, and introduces a number of key concepts and components of the Pelican framework.

The following sections of the reference documentation are relevant for importing data into Pelican:

- Chunker reference
- Data Client reference
- Data Blob reference
- Adapter reference
- Configuration reference

As well as being an introduction, this chapter includes a tutorial to illustrate how to connect to a UDP data stream containing a signal, how to build a very basic but fully-working Pelican pipeline to process the data, and finally how to output the processed data. The tutorial will step through a number of code examples and explain the steps required to implement each component.

First, we introduce some general terminology used in the Pelican framework.

3.2.2 Input Data Types

Pelican distinguishes between two classes of data, namely *Stream Data* and *Service Data* (described below). There can be any number of stream data and service data types, but each one must be handled separately (each will need its own adapter and chunker; these components are described in later sections). Each source of incoming data, either stream or service, must have its own unique identifier in the form of a string.

3.2.2.1 Stream Data

This is an incoming data stream that is expected to be a fairly continuous series of similarly structured data. As it arrives it is automatically associated with any available service data.

3.2.2.2 Service Data

This is relatively static data that is updated occasionally and supports the stream data (e.g. it may describe the incoming stream data format), or status information of instruments collecting the data. There is always a "current" version of any type of service data, and it is this version that is associated with any incoming stream data.

3.2.3 An Example Input Stream

To begin our tutorial, let's suppose that we have a source of input stream data from a UDP socket on port 2001. The UDP packets, in this example, are 1056 bytes long and consist of a 32-byte header followed by 256 samples of 4-byte floating-point values from a digitised time series. For this example, the signal in the time series will be a simple sine wave.

The header contains the following fields:

- Header + 0: A four-byte integer containing the total size of the UDP packet, in bytes (1056).
- Header + 4: A four-byte integer containing the size of the header, in bytes (32).

- Header + 8: A four-byte integer containing the number of samples in the packet (256).
- Header + 12: A four-byte integer containing the size of a single sample, in bytes (4).
- Header + 16: An eight-byte integer containing a monotonically-increasing packet sequence number.

The remainder of the header data is reserved. Each sample represents 10 microseconds of data, so each UDP packet will arrive roughly every 2.56 milliseconds.

3.2.3.1 Emulating Input Stream Data

This data stream can be emulated using the classes provided in the data emulation framework. The EmulatorDriver class takes care of the common, low-level details. It creates a separate thread in which to run the data emulator, and is responsible for writing the data packets to the output device. On construction, the EmulatorDriver takes ownership of a pointer to an AbstractEmulator derived class, which is called repeatedly by the driver to generate the required data packets.

The primary function of the data emulator is to provide a pointer to a block of memory to use when sending data packets. Since we are using a UDP stream, we can use the AbstractUdpEmulator as a base class for our SignalEmulator. Here is the C++ header file containing the class declaration:

```
#ifndef SIGNALEMULATOR_H
#define SIGNALEMULATOR_H

#include "emulator/AbstractUdpEmulator.h"
#include <QtCore/QByteArray>

namespace pelican {

class ConfigNode;

/*
 * This emulator outputs simple packets of real-valued, floating-point,
 * time-series data using a UDP socket. A sine wave is put into the time
 * stream.
 * It should be constructed with a configuration node that contains
 * the number of samples in the packet, the interval between packets in
 * microseconds, and the connection settings.
 *
 * The default values are:
 *
 * <SignalEmulator>
 *     <packet samples="256" interval="2560" />
 *     <signal period="20" />
 *     <connection host="127.0.0.1" port="2001" />
 * </SignalEmulator>
 */
class SignalEmulator : public AbstractUdpEmulator
{
    public:
        // Constructs a new UDP packet emulator.
        SignalEmulator(const ConfigNode& configNode);

        // Destroys the UDP packet emulator.
        ~SignalEmulator() {}

        // Creates a UDP packet.
        void getPacketData(char*& ptr, unsigned long& size);

        // Returns the interval between packets in microseconds.
        unsigned long interval() {return _interval;}

    private:
        unsigned long _counter;  // Packet counter.
        unsigned long long _totalSamples; // The total number of samples sent.
        unsigned long _samples; // The number of samples in the packet.
        unsigned long _interval; // The interval between packets in microsec.
        unsigned long _period; // The number of samples in one period.
        float _omega; // The angular frequency of the signal (from _period).
        QByteArray _packet; // The data packet.
};

} // namespace pelican

#endif // SIGNALEMULATOR_H
```

We simply have to implement the AbstractEmulator::getPacketData() method, which in this case is responsible for writing the correct data into a single UDP packet. Here's the class implementation in the C++ source file:

```
#include "tutorial/SignalEmulator.h"
#include "utility/ConfigNode.h"
#include <cmath>

namespace pelican {

/*
 * Constructs the SignalEmulator.
 * This obtains the relevant configuration parameters.
 */
SignalEmulator::SignalEmulator(const ConfigNode& configNode)
    : AbstractUdpEmulator(configNode)
{
    // Initialise defaults.
    _counter = 0;
    _totalSamples = 0;
    _samples = configNode.getOption("packet", "samples", "256").toULong();
    _interval = configNode.getOption("packet", "interval",
            QString::number(_samples * 10)).toULong(); // Interval in
       micro-sec.
    _period = configNode.getOption("signal", "period", "20").toULong();
    _omega = (2.0 * 3.14159265) / _period; // Angular frequency.

    // Set the packet size in bytes (+32 for header).
    _packet.resize(_samples * sizeof(float) + 32);

    // Set constant parts of packet header data.
    char* ptr = _packet.data();
    *reinterpret_cast<int*>(ptr + 0) = _packet.size(); // Total bytes in
       packet.
    *reinterpret_cast<int*>(ptr + 4) = 32; // Size of the header in bytes.
    *reinterpret_cast<int*>(ptr + 8) = _samples; // Samples in the packet.
    *reinterpret_cast<int*>(ptr + 12) = sizeof(float); // Bytes per sample.
}

/*
 * Creates a packet of UDP signal data containing a sine wave, setting the
 * pointer to the start of the packet, and the size of the packet.
 */
void SignalEmulator::getPacketData(char*& ptr, unsigned long& size)
{
    // Set pointer to the output data.
    ptr = _packet.data();
    size = _packet.size();

    // Set the packet header.
    *reinterpret_cast<long int*>(ptr + 16) = _counter;

    // Fill the packet data.
    char* data = ptr + 32; // Add offset for header.
    for (unsigned i = 0; i < _samples; ++i) {
        float value = sin(((_totalSamples + i) % _period) * _omega);
        reinterpret_cast<float*>(data)[i] = value;
    }

    // Increment counters for next time.
    _counter++;
    _totalSamples += _samples;
}

} // namespace pelican
```

Note that a reference to the emulator's XML configuration node should be supplied to the constructor: we then just need to extract relevant configuration settings using methods on the supplied ConfigNode object. The ConfigNode-::getOption method returns a QString containing the text in the specified tag name, attribute pair. To illustrate what is needed here, the emulator expects an XML snippet like this:

```
<SignalEmulator>
    <packet samples="256" interval="2560" />
    <signal period="20" />
    <connection host="127.0.0.1" port="2001" />
</SignalEmulator>
```

Each time SignalEmulator::getPacketData() is called, the contents of the packet must be updated ready for sending. The function simply updates the header and writes a sine wave into the data section.

This concludes our description of the input data stream emulator.

3.2.4 Chunkers

A *chunker* is the first component of a functioning processing system. Its purpose is to connect to a source of input data (for example, a network socket, or a local file) and turn it into suitable size chunks. Each chunk of data will be processed by one iteration of a single Pelican pipeline. Since the input data rate may be high, a chunker should not, in general, attempt to process, re-order or otherwise interpret the input data stream, but should simply accumulate enough data to form a complete chunk. This may mean that large chunks will need to be assembled from many smaller data packets, if the chunker connects to a network socket. If the data needs to be rearranged for optimal processing, then this rearrangement should be performed by the data adapter (described in a later section).

3.2.4.1 Tutorial Chunker

To connect to our example UDP data stream, we must implement a chunker that inherits the AbstractChunker class interface. For our purposes, let's suppose that our fictional processing pipeline (described in a later section) requires 8192 time-samples to operate on. Each UDP packet from our emulator is 1056 bytes long and will contain 256 time samples, so our chunker will need to accumulate 32 UDP packets, or 33792 bytes, to form a complete chunk of data.

The C++ header file for our SignalChunker is shown below:

```
#ifndef SIGNALCHUNKER_H
#define SIGNALCHUNKER_H

#include "server/AbstractChunker.h"

using namespace pelican;

/*
 * A simple example to demonstrate how to write a data chunker.
 */
class SignalChunker : public AbstractChunker
{
    public:
        // Constructs the chunker.
        SignalChunker(const ConfigNode& config);

        // Creates the input device (usually a socket).
        virtual QIODevice* newDevice();

        // Obtains a chunk of data from the device when data is available.
        virtual void next(QIODevice*);

    private:
        qint64 _chunkSize;
        qint64 _bytesRead;
};

PELICAN_DECLARE_CHUNKER(SignalChunker)

#endif // SIGNALCHUNKER_H
```

The chunker must be declared so that the Pelican framework knows of its existence: use the PELICAN_DECLARE_CHUNKER macro in the header file to do this, supplying the class name as the macro argument. Do not use quotes around the name.

As well as the constructor, we must implement the AbstractChunker::newDevice() method, which creates a new input device (a UDP socket in this case), and the AbstractChunker::next() method, which is called whenever there is new data to read from the device.

Here's the class implementation in the C++ source file:

```
#include "tutorial/SignalChunker.h"
#include "utility/Config.h"
#include <QtNetwork/QUdpSocket>

// Construct the example chunker.
SignalChunker::SignalChunker(const ConfigNode& config) : AbstractChunker(config
      )
{
    // Set chunk size from the configuration.
    // The host, port and data type are set in the base class.
    _chunkSize = config.getOption("data", "chunkSize").toInt();
```

```
}

// Creates a suitable device ready for reading.
QIODevice* SignalChunker::newDevice()
{
    // Return an opened QUdpSocket.
    QUdpSocket* socket = new QUdpSocket;
    socket->bind(QHostAddress(host()), port());

    // Wait for the socket to bind.
    while (socket->state() != QUdpSocket::BoundState) {}
    return socket;
}

// Called whenever there is data available on the device.
void SignalChunker::next(QIODevice* device)
{
    // Get a pointer to the UDP socket.
    QUdpSocket* udpSocket = static_cast<QUdpSocket*>(device);
    _bytesRead = 0;

    // Get writable buffer space for the chunk.
    WritableData writableData = getDataStorage(_chunkSize);
    if (writableData.isValid()) {
        // Get pointer to start of writable memory.
        char* ptr = (char*) (writableData.ptr());

        // Read datagrams for chunk from the UDP socket.
        while (isActive() && _bytesRead < _chunkSize) {
            // Read the datagram, but avoid using pendingDatagramSize().
            if (!udpSocket->hasPendingDatagrams()) {
                // MUST WAIT for the next datagram.
                udpSocket->waitForReadyRead(100);
                continue;
            }
            qint64 maxlen = _chunkSize - _bytesRead;
            qint64 length = udpSocket->readDatagram(ptr + _bytesRead, maxlen);
            if (length > 0)
                _bytesRead += length;
        }
    }

    // Must discard the datagram if there is no available space.
    else {
        udpSocket->readDatagram(0, 0);
    }
}
```

Note that a reference to the XML configuration node for the chunker will be automatically supplied to the constructor from the application's configuration file: we then simply need to extract the relevant configuration settings using methods on the supplied ConfigNode object. The ConfigNode::getOption method returns a QString containing the text in the required tag name and attribute function arguments. To illustrate what is needed here, the chunker expects an XML snippet like this:

```
<SignalChunker>
    <data type="SignalData" chunkSize="33792" />
    <connection host="127.0.0.1" port="2001" />
</SignalChunker>
```

The data type, host name and port number are stored by the AbstractChunker base class, and so do not need to be read again. The contents of the string containing "chunkSize" are converted to an integer value using the QString::toInt() method. This initialisation step only happens once, at program launch, so there is no cost penalty involved here. The attributes are stored in private class variables for later use.

Our SignalChunker::next() method must first call the inherited method AbstractChunker::getDataStorage(). This calls routines in the DataManager and returns a pointer to a block of memory of the required size, wrapped inside a WritableData container. When the WritableData object goes out of scope at the end of the method, the chunk of data it references is automatically placed on a queue so that it can be processed by an available pipeline. The SignalChunker::next() method must therefore accumulate the required number of UDP packets into a chunk before exiting.

After obtaining the block of writable data, we must check that the memory it holds is actually available for writing using WritableData::isValid(): if not (for example, because the buffer has run out of space), the UDP datagram must be discarded. Provided that the memory is valid, we obtain a pointer to it using WritableData::ptr(), and read the data from the socket using the methods on QUdpSocket or QIODevice: A pointer to the socket is passed to the

function, so we can call QUdpSocket::readDatagram(). If there is not enough data to read from the socket, then we must call QIODevice::waitForReadyRead() to suspend the thread and wait for more data to arrive.

This concludes our description of the required SignalChunker. The following section shows how to use the chunker in the next part of the Pelican framework, and how to extract the bytes in the chunk to a form more suitable for processing.

3.3 Importing Data II: The Server, Client and Data Adapters

3.3.1 Introduction

Chunkers, described in the previous section, can be used in Pelican in two main framework configurations, distinguished by whether data is received by a server communicating with one or more data clients or directly by a data client. These framework configurations are designed to make maximum use of the same underlying modules for handling the input data stream but can be configured differently to provide optimisation in different use cases.

The following sections introduce each of these different framework modes and then describe the final step in the data acquisition, namely adapting or de-serialising chunks of data.

For the purposes of the tutorial, please refer to the final section of the getting started guide for how to construct and configure simple binaries making use of the Pelican framework.

3.3.2 The Pelican Server

The Pelican Server provides a framework for managing one or more data streams which are split, buffered and served on request. Data arrives in the server though chunkers and is put in a number of buffers which store it until a request is made by a data client attached to processing pipelines. The server handles requests for data by spawning a thread which will wait until it can satisfy the data requirements for at least one pipeline before proceeding to send data for processing. On sending data, locks are applied so that more than multiple requests can be handled concurrently, each being provided with unique data . In this way the server provides a scalable solution to split incoming data streams to a number of parallel processing pipelines.

3.3.2.1 Attaching chunkers to the server

The Pelican Server requires you to specify one or more chunkers and associate these chunkers to their named data streams.

This is done with the PelicanServer::addStreamChunker() method and PelicanServer::addServiceChunker() methods. As the names suggest, the former is used to associate chunkers with data streams that are to be treated as stream data, and the latter for data that is to be treated as service data.

3.3.3 Data Clients

A data-client is an implementation of the AbstractDataClient interface, which is the primary interface used by the processing pipeline for accessing data when it needs it.

Pelican provides three different data-client implementations that you can configure to your needs:

- The PelicanServerClient connects to a Pelican server and so only requires an adapter (the chunker being in the server itself).
- The DirectStreamDataClient connects directly to the incoming data stream and so needs both an adapter and a chunker.
- The FileDataClient provides data directly from one or more files though an adapter interface. This data client can be extremely useful for testing.

These data clients, described in more detail in the reference documentation.

3.3.3.1 Specifying the Data Client

The PelicanApplication needs to know which data client will be used for importing data. You specify the class by calling the PipelineApplication::setDataClient() method, supplying the name of the data client's class as a string.

e.g. for the DirectDataClientExample below you would use:

```
PipelineApplication* app; // initialised application
app->setDataClient("DirectDataClientExample");
```

3.3.3.2 Installing Chunkers into the DirectStreamDataClient

Chunkers are attached to the direct stream client by creating a derived class which inherits DirectStreamDataClient, and using the DirectStreamDataClient::addChunker to specify the data type, the chunker type, and chunker name as arguments.

For example if we want to attach two chunkers of the same type to two differently named data streams:

```
#ifndef DIRECTDATACLIENTEXAMPLE_H
#define DIRECTDATACLIENTEXAMPLE_H


#include "core/DirectStreamDataClient.h"

using namespace pelican;

class DirectDataClientExample : public DirectStreamDataClient
{
    public:
        DirectDataClientExample( const ConfigNode& configNode,
                 const DataTypes& types, const Config* config );

        ~DirectDataClientExample();

    private:
};

PELICAN_DECLARE_CLIENT(DirectDataClientExample)

#endif // DIRECTDATACLIENTEXAMPLE_H


#include "reference/DirectDataClientExample.h"
#include "reference/ChunkerExample.h"


using namespace pelican;

DirectDataClientExample::DirectDataClientExample(const ConfigNode& configNode,
                const DataTypes& types, const Config* config)
    : DirectStreamDataClient(configNode, types, config)
{
    addChunker("Stream1", "ChunkerExample", "Stream1ConfigName");
    addChunker("Stream2", "ChunkerExample", "Stream2ConfigName");
}

DirectDataClientExample::~DirectDataClientExample()
{
}
```

The chunkers will be initialised with the relevant XML node derived from the chunker class name and name string specified as the second and third arguments of the *addChunker* method.

Currently there is no way of attaching chunkers via the XML configuration however this is likely to be available in the next release.

3.3.3.3 Installing an Adapter into a DataClient

Adapters, described below, are attached to the data client via the XML configuration common to all data clients which describe the mapping of the various data types handled by the client and their associated adapters. This mapping is specified by a *data* tag with two attributes; a *type* attribute holding the name of the data chunk / stream to be adapted, and an *adapter* attribute giving the class name of the adapter to be attached. Construction and ataching adapters is then handled automatically by the Pelican framework (DataClientFactory).

For example:

```
    <MyDataClient>
        <data type="MyData" adapter="AdapterForMyData"/>
        <data type="MyOtherData" adapter="AdapterForMyOtherData"/>
    </MyDataClient>
```

Currently there is no easy way to assign adapters programmatically although this feature is likely to be available in the next release.

3.3.4 Adapters

Adapters are the final components of the data-import chain, and provide a mechanism to convert chunks of raw binary data into the data members of a Pelican data-blob (a specialised C++ container for holding data used by the Pelican pipeline; see below). The most basic function of an adapter is to de-serialise chunks of data, although re-ordering and re-factoring of the data to a form that is convenient for subsequent pipeline processing may also be carried out. Pelican currently provides support for two categories of adapters, distinguished by the type of input data chunks they are expected to process: these are stream data adapters and service data adapters, which operate on the relevant data types.

Adapters can be configured, if required, using parameters specified in the XML configuration file. An adapter must inherit either the AbstractStreamAdapter or the AbstractServiceAdapter class interface, depending on its type.

3.3.4.1 Data-Blobs

A Pelican *data-blob* is a well-structured representation of data that has been arranged for easy, optimal processing: the output from an adapter is a well-structured representation of the data held in a chunk, which is ready to be processed by a pipeline. Data-blobs may contain arrays, blocks of memory and/or other metadata, and should provide methods to interact with that data. One of their main functions is to act as an interface between pipeline modules.

3.3.4.2 Tutorial Data-Blob

For the purpose of our tutorial, we need a type of DataBlob that can hold the required time-series data in a chunk. This is implemented in the SignalData data-blob, shown below, as a simple contiguous float array (a standard C++ vector). Public methods allow this array to be resized, and should be used to obtain a pointer to the start of the data so that elements can be accessed efficiently.

```
#ifndef SIGNALDATA_H
#define SIGNALDATA_H

#include "data/DataBlob.h"
#include <vector>

using namespace pelican;

/*
 * This data blob holds an array of floating-point time-stream data.
 */
class SignalData : public DataBlob
{
    public:
        // Constructs a signal data blob.
        SignalData() : DataBlob("SignalData") {}

        // Returns a const pointer to the start of the data.
        const float* ptr() const {return (_data.size() > 0 ? &_data[0] : NULL);
      }

        // Returns a pointer to the start of the data.
        float* ptr() { return (_data.size() > 0 ? &_data[0] : NULL); }

        // Resizes the data store provided by the data blob.
        void resize(unsigned length) { _data.resize(length); }

        // Returns the size of the data.
        unsigned size() const { return _data.size(); }

    private:
        std::vector<float> _data; // The actual data array.
};

PELICAN_DECLARE_DATABLOB(SignalData)

#endif // SIGNALDATA_H
```

The data-blob must be declared so that the Pelican framework knows of its existence: use the PELICAN_DECLARE_DATABLOB macro in the header file to do this, supplying the class name as the macro argument. Do not use quotes around the name.

The serialise and deserialise methods must be reimplemented for the data output stage, and are described in a later section.

This concludes our description of the required SignalData data-blob.

3.3.4.3 Tutorial Adapter

As a type of AbstractStreamAdapter, our SignalDataAdapter must convert the data held in a chunk into a SignalData data-blob. The adapter must process the chunk, removing all headers from the UDP packets it contains, and reinterpret the contents of the data section as 32-bit floating-point numbers.

The C++ header file for our SignalDataAdapter is shown below:

```
#ifndef SIGNALDATAADAPTER_H
#define SIGNALDATAADAPTER_H

#include "core/AbstractStreamAdapter.h"

using namespace pelican;

/*
 * Adapter to convert chunks of signal stream data into a SignalData data-blob.
 */
class SignalDataAdapter : public AbstractStreamAdapter
{
    public:
        // Constructs the adapter.
        SignalDataAdapter(const ConfigNode& config);

        // Method to deserialise chunks of memory provided by the I/O device.
        void deserialise(QIODevice* device);

    private:
        static const unsigned _headerSize = 32;
        unsigned _samplesPerPacket;
        unsigned _packetSize;
};

PELICAN_DECLARE_ADAPTER(SignalDataAdapter)

#endif // SIGNALDATAADAPTER_H
```

The adapter must be declared so that the Pelican framework knows of its existence: use the PELICAN_DECLARE_ADAPTER macro in the header file to do this, supplying the class name as the macro argument. Do not use quotes around the name.

As well as the constructor, we must implement the AbstractAdapter::deserialise() method, which must deserialise the contents of the chunk into the required data blob.

Here's the class implementation in the C++ source file:

```
#include "tutorial/SignalDataAdapter.h"
#include "tutorial/SignalData.h"

// Construct the signal data adapter.
SignalDataAdapter::SignalDataAdapter(const ConfigNode& config)
    : AbstractStreamAdapter(config)
{
    // Read the configuration using configuration node utility methods.
    _samplesPerPacket = config.getOption("packet", "samples").toUInt();

    // Set up the packet data.
    _packetSize = _headerSize + _samplesPerPacket * sizeof(float);
}

// Called to de-serialise a chunk of data from the input device.
void SignalDataAdapter::deserialise(QIODevice* device)
{
    // A pointer to the data blob to fill should be obtained by calling the
    // dataBlob() inherited method. This returns a pointer to an
    // abstract DataBlob, which should be cast to the appropriate type.
    SignalData* blob = (SignalData*) dataBlob();
```

```
    // Set the size of the data blob to fill.
    // The chunk size is obtained by calling the chunkSize() inherited method.
    unsigned packets = chunkSize() / _packetSize;
    blob->resize(packets * _samplesPerPacket);

    // Create a temporary buffer to read out the packet headers, and
    // get the pointer to the data array in the data blob being filled.
    char headerData[_headerSize];
    char* data = (char*) blob->ptr();

    // Loop over the UDP packets in the chunk.
    unsigned bytesRead = 0;
    for (unsigned p = 0; p < packets; ++p)
    {
        // Ensure there is enough data to read from the device.
        while (device->bytesAvailable() < _packetSize)
            device->waitForReadyRead(-1);

        // Read the packet header from the input device and dump it.
        device->read(headerData, _headerSize);

        // Read the packet data from the input device into the data blob.
        bytesRead += device->read(data + bytesRead, _packetSize - _headerSize);
    }
}
```

A reference to the XML configuration node for the adapter will be automatically supplied to the constructor from the application's configuration file: we then simply need to extract the relevant configuration settings using methods on the supplied ConfigNode object. The ConfigNode::getOption method returns a QString containing the text in the required tagname and attribute. To illustrate what is needed here, the adapter expects an XML snippet like this:

```
<SignalDataAdapter>
    <packet samples="256" />
</SignalDataAdapter>
```

The contents of the string containing the number of samples per packet are converted to an integer value using the QString::toInt() method, and stored in a private class variable for later use. Note that we did not strictly need to supply the number of samples per packet in this example, since the relevant information is contained in the header section of the UDP packet data; however, this serves to illustrate the steps that may be required to configure a more complicated adapter.

The AbstractAdapter is aware of the data blob that must be filled, so in our SignalDataAdapter::deserialise() routine, we must obtain a pointer to this data blob by calling the AbstractAdapter::dataBlob() inherited method. The chunk size is also available, and can be retrieved using the AbstractAdapter::chunkSize() inherited method.

The remainder of the deserialise routine simply loops over the UDP packets in the chunk, discarding the headers in each case, and reading the data sections into the correct parts of the SignalData data-blob.

Note

It is important, especially when the input device is a TCP socket, to check that sufficient data is available before trying to read it. For this reason we have included a call to QIODevice::waitForReadyRead() if there is not enough data available on the device.

This concludes our description of the required SignalDataAdapter. The following section shows how to process this data by creating a simple module and inserting it into a processing pipeline.

3.4 Processing Data: Pipelines and Modules

3.4.1 Introduction

The following sections of the reference documentation are relevant for processing data using Pelican:

- Pipeline reference
- Module reference
- Data Blob reference
- Writing main()

3.4.2 Overview

This section will explain how to build a data processing pipeline and compile it into a working binary that can be launched on a data stream.

We continue the example started in the previous section. Having acquired the signal data from the network, transforming it from UDP packets using the SignalChunker and converting it into a SignalData data-blob using the SignalDataAdapter, we are ready to do some real work with the data.

First, we describe the details of the required pipeline module.

3.4.3 Pipeline Modules

Modules are the components of the Pelican framework that perform intensive computational operations on the data flowing through them. Normally, several modules would be run in serial to build up a processing pipeline. A module may contain anything: it can be as simple or as complex as needed to perform the task required of it, and use (for example) multiple threads, OpenMP, CUDA or OpenCL (if available) to help improve performance.

Data blobs (Pelican data structures) are used to pass data between pipeline modules. Depending on requirements, new types of data blob may need to be implemented to work with a new module.

Pipeline modules can be configured using parameters specified in the XML configuration file. To create a new module, inherit the AbstractModule class interface.

3.4.3.1 Tutorial Module

Since this is not a tutorial on signal processing, we will construct a Pelican pipeline and a pipeline module to do a trivially simple operation on the signal data. A real-world module might do something useful like produce a spectrum from the time-series data; however, we will simply amplify the signal by multiplying all the sample values in the data-blob by a constant (but configurable) factor. The amplified data will be output from the module using another SignalData data-blob created in the pipeline itself.

The code below shows the details of the SignalAmplifier pipeline module, which is a type of AbstractModule. Here's the C++ header file:

```
#ifndef SIGNALAMPLIFIER_H
#define SIGNALAMPLIFIER_H

#include "core/AbstractModule.h"

using namespace pelican;

class SignalData;

/*
 * A simple example to demonstrate how to write a pipeline module.
 */
class SignalAmplifier : public AbstractModule
{
```

```
    public:
        // Constructs the module.
        SignalAmplifier(const ConfigNode& config);

        // Runs the module.
        void run(const SignalData* input, SignalData* output);

    private:
        float gain;
};

PELICAN_DECLARE_MODULE(SignalAmplifier)

#endif // SIGNALAMPLIFIER_H
```

The module must be declared so that the Pelican framework knows of its existence: use the PELICAN_DECLARE-_MODULE preprocessor macro in the header file to do this, supplying the class name as the macro argument. Do not use quotes around the name.

Apart from the constructor, there are no abstract methods that must be implemented on the module; however, by convention, a run() method is defined and called by the pipeline when the module must process a chunk of data, and pointers to the data blobs to use for input and output are provided as function arguments.

Here's the class implementation in the C++ source file:

```
#include "tutorial/SignalAmplifier.h"
#include "tutorial/SignalData.h"
#include "utility/Config.h"

// Construct the example module.
SignalAmplifier::SignalAmplifier(const ConfigNode& config)
    : AbstractModule(config)
{
    // Set amplifier gain from the XML configuration.
    gain = config.getOption("gain", "value").toDouble();
}

// Runs the module.
void SignalAmplifier::run(const SignalData* input, SignalData* output)
{
    // Ensure the output storage data is big enough.
    unsigned nPts = input->size();
    if (output->size() != nPts)
        output->resize(nPts);

    // Get pointers to the memory to use from the data blobs.
    const float* in = input->ptr();
    float* out = output->ptr();

    // Perform the operation.
    for (unsigned i = 0; i < nPts; ++i) {
        out[i] = gain * in[i];
    }
}
```

Note that a reference to the XML configuration node for the module will be automatically supplied to the module's constructor: we then simply need to extract the relevant configuration settings using methods on the supplied Config-Node object. The ConfigNode::getOption method returns a QString containing the text in the supplied tagname and attribute. To illustrate what is needed here, the module expects an XML snippet like this:

```
<SignalAmplifier>
    <gain value="2.5"/>
</SignalAmplifier>
```

The contents of the string are then converted to a floating point value using the QString::toDouble() method. This initialisation step only happens once, at program launch, so there is no cost penalty involved here. The value of the gain attribute is stored in a private class variable for later use when the module is run.

The SignalAmplifier::run() method will be called by the pipeline whenever there is data to process. It is called with pointers to the input and output data blobs as function arguments, and first checks that the output data blob has sufficient capacity to hold the amplified signal: if not, then it is resized.

Pointers to the input and output memory blocks are obtained, and the for-loop iterates over the time samples in the signal, multiplying the input values by the configured gain.

This completes our description of the SignalAmplifier module.

3.4.4 The Pipeline

Pelican pipelines act as containers for the data processing framework. A pipeline is simply a C++ class that defines the operations performed on a single chunk of stream data, and is run repeatedly by the Pelican framework whenever there is a new chunk of data available to be processed. The data processing itself actually happens within pipeline modules.

Pipelines work by pulling data to process whenever they become available. Since a chunk is passed to a pipeline only when it can process the data, multiple concurrent pipelines that use the Pelican server will be load-balanced automatically, even if they run on different types of hardware.

Since it acts only as a container, a pipeline cannot be configured at run-time using the XML configuration file. Instead, pipelines must be written and compiled as C++ objects, which are defined by inheriting the AbstractPipeline class interface. Pipelines initialise themselves by creating the modules and data blobs that they require, and by requesting the input data types from the data client. A pipeline defines the order in which its modules are run, and is responsible for supplying them with the correct data. If necessary, the pipeline must also specify the data to send to the output streamer.

3.4.4.1 Tutorial Pipeline

We now describe the SignalProcessingPipeline that is used to contain the SignalAmplifier pipeline module. Typically, pipelines will contain multiple modules, but in this example, we use only one. Pipelines must inherit the AbstractPipeline Pelican class, which defines the interface required. Here's the C++ header file:

```
#ifndef SIGNALPROCESSINGPIPELINE_H
#define SIGNALPROCESSINGPIPELINE_H

#include "core/AbstractPipeline.h"

using namespace pelican;

class SignalData;
class SignalAmplifier;

class SignalProcessingPipeline : public AbstractPipeline
{
    public:
        // Constructor.
        SignalProcessingPipeline();

        // Destructor
        ~SignalProcessingPipeline();

        // Initialises the pipeline.
        void init();

        // Defines one iteration of the pipeline.
        void run(QHash<QString, DataBlob*>& remoteData);

    private:
        // Module pointers.
        SignalAmplifier* amplifier;

        // Local data blob pointers.
        SignalData* outputData;

        unsigned long counter;
};

#endif // SIGNALPROCESSINGPIPELINE_H
```

Note that pipelines cannot be configured using XML parameters – that functionality can only be in pipeline modules.

There are two abstract methods that must be implemented here: SignalProcessingPipeline::init() is called by the Pelican framework on initialisation to set up the pipeline, and SignalProcessingPipeline::run() is called whenever there is data to be processed.

Here's the class implementation in the C++ source file:

```
#include "tutorial/SignalProcessingPipeline.h"
#include "tutorial/SignalAmplifier.h"
#include "tutorial/SignalData.h"
```

```
#include <iostream>

// The constructor. It is good practice to initialise any pointer
// members to zero.
SignalProcessingPipeline::SignalProcessingPipeline()
    : AbstractPipeline(), amplifier(0), outputData(0), counter(0)
{
}

// The destructor must clean up and created modules and
// any local DataBlob's created.
SignalProcessingPipeline::~SignalProcessingPipeline()
{
    delete amplifier;
    delete outputData;
}

// Initialises the pipeline, creating required modules and data blobs,
// and requesting remote data.
void SignalProcessingPipeline::init()
{
    // Create the pipeline modules and any local data blobs.
    amplifier = (SignalAmplifier*) createModule("SignalAmplifier");
    outputData = (SignalData*) createBlob("SignalData");

    // Request remote data.
    requestRemoteData("SignalData");
}

// Defines a single iteration of the pipeline.
void SignalProcessingPipeline::run(QHash<QString, DataBlob*>& remoteData)
{
    // Get pointers to the remote data blob(s) from the supplied hash.
    SignalData* inputData = (SignalData*) remoteData["SignalData"];

    // Output the input data.
    dataOutput(inputData, "pre");

    // Run each module as required.
    amplifier->run(inputData, outputData);

    // Output the processed data.
    dataOutput(outputData, "post");

    if (counter%10 == 0)
        std::cout << counter << " Chunks processed." << std::endl;

    counter++;
}
```

The SignalProcessingPipeline::init() method must do three things:

- Create the required pipeline modules by calling the inherited AbstractPipeline::createModule() method, supplying the class name of the module as a string.
- Create any required data blobs that are local to the pipeline (e.g. for output data) by calling the inherited AbstractPipeline::createBlob() method, supplying the class name of the data blob as a string. You must delete these when you have finished with them, usually in the destructor.
- Request the input data to be supplied to the pipeline by calling the inherited AbstractPipeline::requestRemoteData() method as many times as necessary, supplying the data-blob class name(s) as a string.

Finally, the SignalProcessingPipeline::run() method defines the order in which modules are run by the pipeline. The function is passed a hash containing pointers to the requested data-blobs, and these pointers should be extracted by looking them up by name in the hash using the square-bracket [] dereference operator.

Pointers to the pipeline modules are stored as class variables, so the modules can be launched and be passed pointers to the appropriate data blobs for processing.

Data products can be output from the pipeline at any stage by calling the AbstractPipeline::dataOutput() method, and this is described more in the next section.

3.5 Data Output

3.5.1 Introduction

The following sections of the reference documentation are relevant for exporting data from Pelican:

- Data Blob reference
- Output streamer reference

3.5.2 Data Output from a Pipeline

In simple cases, you just call the AbstractPipeline::dataOutput() method, and supply a pointer to the data blob that must be sent:

```
MyBlob* data;
dataOutput(data);
```

This will push data onto an output stream named after the data type ("MyBlob" in this case). The AbstractPipeline-::dataOutput() method can take a string as a second argument to assign data to a stream with an alternative name. This can be useful where there you are exporting multiple streams of the same data type.

3.5.3 Data-Blob Output Requirements

It is important to note that any DataBlob destined for output should reimplement the DataBlob::serialise() and Data-Blob::deserialise() methods. The serialise routine should write the complete contents of the data blob to the output device provided in the arument. The deserialise method should do the reverse, reading the complete contents of the blob from the input device. Note that the byte order of the machine that the serialise() method was called is also provided to help you interpret binary streams if necessary.

3.5.3.1 Tutorial Data Blob Output

The SignalData::serialise() and SignalData::deserialise() methods for our SignalData data-blob are shown in the C++ source file, below:

```
#include "tutorial/SignalData.h"
#include <QtCore/QDataStream>

// Serialises the data blob.
void SignalData::serialise(QIODevice& out) const
{
    QDataStream stream(&out);

    // Write the number of samples in the time series.
    unsigned samples = size();
    stream << samples;

    // Write the data to the output device.
    const float* data = ptr();
    for (unsigned i = 0; i < samples; ++i) stream << data[i];
}

// Deserialises the data blob.
void SignalData::deserialise(QIODevice& in, QSysInfo::Endian)
{
    QDataStream stream(&in);

    // Read the number of samples in the time series.
    unsigned samples = 0;
    stream >> samples;

    // Set the size of the blob and read the data into it.
    resize(samples);
    float* data = ptr();
    for (unsigned i = 0; i < samples; ++i) stream >> data[i];
}
```

We use a QDataStream to be endian-neutral, so we don't actually need use the Endian type argument of the deserialise routine: this provides the endianness of the system that serialised the blob.

3.5.4 The Output Stream Manager

The actual destination of the data is controlled by the OutputStreamManager object. This can be configured either through the configuration file, or you can manipulate the object directly.

Pelican comes with two output streamer modules by defualt (described in more detail in the reference manual). Both assume that you have implemented the serialise() and deserialise methods. However, this may be insufficient for your needs if you have a custom file format, or a specific database to fill. In these cases you will need to write your own output streamer, which we will give an example of after discussing those provided by Pelican.

3.5.4.1 The DataBlobFile

This will store a binary format file on the local machine. There is a *DataBlobFileReader* available for reading this format and extracting its contents as DataBlobs for further processing as required.

3.5.4.2 The PelicanTCPBlobServer

The PelicanTCPBlobServer allows you to stream data to one or more other machines using a network TCP connection. As any processing done in an output streamer is done serially within the pipeline processing, and so any latencies, e.g disk writes, will slow down your processing. If this will be a problem, it is better to export the data via the PelicanTCPBlobServer and process the streams elsewhere. See user_dataRelay the data relay section for more information.

3.5.4.3 Tutorial Output

We take the example pipeline from previous sections to illustrate how the data output system works. We will output the SignalData data-blob before and after processing to different streams, called "pre" and "post". We will then connect different modules to these streams to show how to redirect the data. The first thing to do is to instruct the pipeline to output data at the required points by calling AbstractPipeline::dataOutput(), as shown. Note that we must specify unique stream names, since both outputs use a data-blob of type SignalData.

```
void SignalProcessingPipeline::run(QHash<QString, DataBlob*>& data)
{
    // Get pointers to the remote data blob(s) from the supplied hash.
    SignalData* inputData = (SignalData*) remoteData["SignalData"];

    // Output the input data.
    dataOutput(inputData, "pre");

    // Run each module as required.
    // ...

    // Output the processed data.
    dataOutput(outputData, "post");
}
```

3.5.4.3.1 Configuring the Output Stream Manager

So now we have two data streams which we can direct wherever required using the output manager:

- The unprocessed data (the "pre" stream) will be directed to a comma-separated value (CSV) file on the local machine using our own custom output streamer.
- Similarly, we direct the processed data stream ("post") to a different CSV file using another instance of our own output streamer.
- We also save a binary format file of the "post" data stream using the default DataBlobFile

- Finally, we will also direct both streams to the standard PelicanTCPBlobServer for export to other clients.

The local CSV files will be written using our custom OutputStreamerExample object described in the Output Streamer Reference. Please see that page for example code that describes how to implement the required output streamer.

The `output` section of our configuration file will look like this:

```
<output>
    <streamers>
        <PelicanTCPBlobServer active="true">
             <connection port="1234" />
        </PelicanTCPBlobServer>
        <DataBlobFile>
            <file name="datablob.dblob" type="heterogeneous">
        </DataBlobFile>
        <OutputStreamExample name="precsv" active="true">
             <file name="pre.csv" />
        </OutputStreamExample>
        <OutputStreamExample name="postcsv" active="true">
             <file name="post.csv" />
        </OutputStreamExample>
    </streamers>
    <dataStreams>
        <stream name="all" listeners="PelicanTCPBlobServer" />
        <stream name="post" listeners="DataBlobFile" />
        <stream name="post" listeners="postcsv" />
        <stream name="pre" listeners="precsv" />
    </dataStreams>
</output>
```

The `streamers` section describes the configuration for each output streamer. Its presence in the XML will cause the OutputStreamManager to attempt to instantiate an object of that type (unless the `active` attribute is set to false). Where there are more than two objects of the same type, you must provide a `name` attribute to disambiguate them.

The `dataStreams` section is used to map the streams to these objects. The `listeners` attribute is a comma separated list of the names of the streamers that should be associated with the named stream. Note the special stream name "all" which will cause all streams to be piped to the listed listeners.

3.5.5 Reading from the PelicanTCPBlobServer

By piping the streams to the PelicanTCPBlobServer, we have the ability to connect from elsewhere to read any of the data streams we are interested in. Pelican provides a client (the DataBlobClient object) to connect to this server and subscribe to any number of streams, reconstructing the DataBlob objects as they are passed over the TCP stream. Use of the DataBlobClient is outside the scope of this tutorial, but it is described in the Pelican API reference.

3.6 Tutorial Conclusion

In this section we conclude the tutorial and put all the examples together to create applications that use the Pelican library.

3.6.1 The Emulator Binary

First, we need a program to output UDP packets to act as an emulator, and provide our test pipeline with some data to process. The following code is a short program to perform this function:

```
#include "emulator/EmulatorDriver.h"
#include "utility/ConfigNode.h"
#include "tutorial/SignalEmulator.h"
#include <QtCore/QCoreApplication>
#include <QtCore/QString>

using namespace pelican;

int main(int argc, char* argv[])
{
    // Create a QCoreApplication.
    QCoreApplication app(argc, argv);

    // Create the emulator's XML configuration node from a string.
    ConfigNode xmlNode(
            "<SignalEmulator>"
            "    <packet samples=\"256\" interval=\"2560\" />"
            "    <signal period=\"20\" />"
            "    <connection host=\"127.0.0.1\" port=\"2001\" />"
            "</SignalEmulator>"
    );

    // Create the emulator and the emulator driver.
    EmulatorDriver driver(new SignalEmulator(xmlNode));
    return app.exec();
}
```

3.6.1.1 Running the Emulator

The emulator can be started simply by running the `./signalEmulator` binary.

3.6.2 The Server Binary

Next, we need to compile a Pelican server to host our chunker, and to buffer the data until a pipeline is ready to process it. The following code is all that is needed to set up our server:

```
#include "server/PelicanServer.h"
#include "comms/PelicanProtocol.h"
#include "utility/Config.h"

#include "tutorial/SignalChunker.h"

#include <QtCore/QCoreApplication>
#include <iostream>

using namespace pelican;

int main(int argc, char ** argv)
{
    // 1. Create a QCoreApplication.
    QCoreApplication app(argc, argv);

    // 2. Create a Pelican configuration object (this assumes that a Pelican
    // configuration XML file is supplied as the first command line argument)
    if (argc != 2) {
        std::cerr << "Please supply an XML config file." << std::endl;
        return 0;
    }
    QString configFile(argv[1]);
    Config config(configFile);

    try {
```

```
        // 3. Create a Pelican server.
        PelicanServer server(&config);

        // 4. Attach the chunker to server.
        server.addStreamChunker("SignalChunker");

        // 5. Create a communication protocol object and attach it to the
       server
        // on port 2000.
        AbstractProtocol *protocol =  new PelicanProtocol;
        server.addProtocol(protocol, 2000);

        // Start the server.
        server.start();

        // 6. When the server is ready enter the QCoreApplication event loop.
        while (!server.isReady()) {}
        return app.exec();
    }

    // 7. Catch any error messages from Pelican.
    catch (const QString& err)
    {
        std::cerr << "Error: " << err.toStdString() << std::endl;
        return 1;
    }
}
```

3.6.2.1 Server XML

Components within the server need to be configured, so the following XML file must be supplied to the server binary:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pelican>

<configuration version="1.0">
    <server>
        <buffers>
            <SignalData>
                <buffer maxSize="50000000" maxChunkSize="50000000" />
            </SignalData>
        </buffers>

        <chunkers>
            <SignalChunker>
                <data type="SignalData" chunkSize="33792" />
                <connection host="127.0.0.1" port="2001" />
            </SignalChunker>
        </chunkers>
    </server>
</configuration>
```

3.6.2.2 Running the Server

The server can be started with the following command, specifying the correct path to the server configuration file:

```
./signalServer tutorial/data/serverConfig.xml
```

3.6.3 The Pipeline Binary

Finally, we need to compile a Pelican pipeline to process our data. The following code is all that is needed to set up the pipeline:

```
#include "core/PipelineApplication.h"
#include "tutorial/SignalProcessingPipeline.h"
#include <QtCore/QCoreApplication>

// Include any headers that are referenced by name.
#include "tutorial/OutputStreamExample.h"
#include "tutorial/SignalDataAdapter.h"
```

```
int main(int argc, char* argv[])
{
    // Create a QCoreApplication.
    QCoreApplication app(argc, argv);

    try {
        // Create a PipelineApplication.
        PipelineApplication pApp(argc, argv);

        // Register the pipelines that can run.
        pApp.registerPipeline(new SignalProcessingPipeline);

        // Set the data client.
        pApp.setDataClient("PelicanServerClient");

        // Start the pipeline driver.
        pApp.start();
    }

    // Catch any error messages from Pelican.
    catch (const QString& err) {
        std::cerr << "Error: " << err.toStdString() << std::endl;
    }

    return 0;
}
```

3.6.3.1 Pipeline XML

We must configure the pipeline module, the data client, the data adapter and the output streams, so the following XML file must be provided for the pipeline binary:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pelican>

<configuration version="1.0">
    <pipeline>
        <clients>
            <PelicanServerClient>
                <server host="127.0.0.1" port="2000" />
                <data type="SignalData" adapter="SignalDataAdapter" />
            </PelicanServerClient>
        </clients>

        <adapters>
            <SignalDataAdapter>
                <packet samples="256" />
            </SignalDataAdapter>
        </adapters>

        <modules>
            <SignalAmplifier>
                <gain value="2.5"/>
            </SignalAmplifier>
        </modules>

        <output>
            <streamers>
                <PelicanTCPBlobServer active="false">
                    <connection port="1234" />
                </PelicanTCPBlobServer>
                <OutputStreamExample name="precsv" active="true">
                    <file name="pre.csv" />
                </OutputStreamExample>
                <OutputStreamExample name="postcsv" active="true">
                    <file name="post.csv" />
                </OutputStreamExample>
            </streamers>
            <dataStreams>
                <stream name="all" listeners="PelicanTCPBlobServer" />
                <stream name="post" listeners="postcsv" />
                <stream name="pre" listeners="precsv" />
            </dataStreams>
        </output>
```

```
    </pipeline>
</configuration>
```

3.6.3.2 Running the Pipeline

The pipeline can be started with the following command, specifying the correct path to the pipeline configuration file:

```
./signalPipeline tutorial/data/pipelineConfig.xml
```

3.6.4 Data Output

The following figure shows example output from the signal processing pipeline constructed for our tutorial:

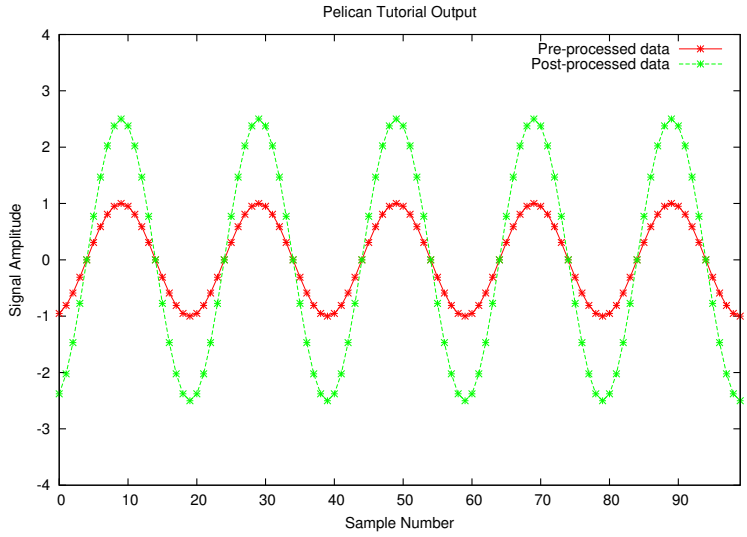

Figure 3.1: Pelican Output

The graph was created by plotting the contents of the CSV files using gnuplot with the following script:

```
# Gnuplot script for plotting Pelican tutorial data.

# Set axis properties, title and labels.
set xlabel "Sample Number"
set ylabel "Signal Amplitude"
set title "Pelican Tutorial Output"
set yrange [-4:4]

# Load the data.
set datafile separator ","
plot [0:99] "pre.csv" matrix every 1:999:0:0 title "Pre-processed data" with lp pt 3 lc 1 lt 1
replot "post.csv" matrix every 1:999:0:0 title "Post-processed data" with lp pt 3 lc 2 lt 1

# Set PostScript output.
set terminal postscript enhanced colour dashed lw 1 "Helvetica" 14
set output "pelican-output.ps"
replot

# Set PNG output.
set terminal png
set output "pelican-output.png"
replot

# Return to the x11 display.
```

```
set terminal x11 persist
replot
```

The script simply plots the contents of the first row of the input data files, truncating the x-axis to 100 values. Since our input sine wave had a period of 20 samples, this plot only shows five complete periods. Other rows of the input data can be selected by changing the last '0' of the 'plot every' command

As expected, the input (pre-processed) data is a sine-wave of amplitude 1.0, and the signal in the output (post-processed) data is 2.5 times larger.

Chapter 4

Pelican Reference

This section provides a detailed reference guide for specific Pelican components, how to use them and how to create new ones.

- Data Chunkers
- Data Clients
- Data Adapters
- Data Blobs
- Pipeline Modules
- Pipelines
- Output Streamers
- Configuration Files
- Data Emulation
- Building Pelican Binaries
- Building the Pelican library
- Testing Utilities

4.1 Data Chunkers

4.1.1 Introduction

The function of the chunker is to take an incoming data stream and turn it into suitable size chunks that can be fed into the data adapter.

Note

Each chunk should contain all the data required for a single iteration of the pipeline. This may mean that chunks will need to be assembled from many smaller network packets, if the chunker connects to a UDP stream.

4.1.2 Overview

To create a new data chunker:

- All chunkers must inherit the `AbstractChunker` class interface.
- In the derived class, implement the `newDevice()` method. This method must create and return an open `QIODevice` ready for input (for example, a `QUdpSocket`). The base class will take ownership of the device, so it must not be deleted explicitly.
- In the derived class, implement the `next()` method. This method will be called automatically whenever data is available on the device created in `newDevice()`. You must check and read off the number of bytes available from the device, and store them in a local buffer before returning. When enough data has arrived to make a chunk, call `getDataStorage()` to return a `WritableData` object. Finally, use WritableData::write() to save the chunk of data into the data manager.
- Chunkers must register their existence with the chunker factory. Use the `PELICAN_DECLARE_CHUNKER()` macro under the class definition in the chunker's header file to register the chunker, supplying the name of the chunker class as the macro argument. Do not use quotes around the name.

Note

All pending data must be read from the device when next() is called, otherwise the method will not be called again when more data is available.

4.1.3 Configuration

All chunkers must be supplied with a configuration node in their constructors. The configuration node resides in the `chunkers` section of the XML file, and has a tag name the same as the chunker class name.

The base class sets the hostname and port from the `connection` tag, which must have `host` and `port` attributes. The base class also sets up the data type that the chunker provides, so that the data manager returns a block of memory from the correct buffer. Use the `data` tag with the `type` attribute, as in the following example.

4.1.4 Built In Chunkers

4.1.4.1 FileChunker

This chunker will monitor a file on the local file system. Every time the file is updated it will transfer the entire contents of the file into a new data chunk ready for serving.

example configuration:

```
<FileChunker file="/path/to/myfile" />
```

4.1.5 Example

In the following, a new chunker is created to read data from a UDP socket. The XML contents of its configuration node are:

```
<ChunkerExample>
    <connection host="127.0.0.1" port="2001"/>
    <data type="VisibilityData" chunkSize="8192"/>
</ChunkerExample>
```

The class definition is:

```
#ifndef CHUNKEREXAMPLE_H
#define CHUNKEREXAMPLE_H

#include "server/AbstractChunker.h"

using namespace pelican;

class ChunkerExample : public AbstractChunker
{
    public:
        ChunkerExample(const ConfigNode& config);
        virtual QIODevice* newDevice();
        virtual void next(QIODevice*);

    private:
        qint64 _chunkSize;
        qint64 _bytesRead;
};

// Register the chunker.
PELICAN_DECLARE_CHUNKER(ChunkerExample)

#endif // CHUNKEREXAMPLE_H
```

and the class implementation is:

```
#include "reference/ChunkerExample.h"
#include "utility/Config.h"
#include <QtNetwork/QUdpSocket>

// Construct the example chunker.
ChunkerExample::ChunkerExample(const ConfigNode& config)
    : AbstractChunker(config)
{
    // Set chunk size from the configuration.
    // The host, port and data type are set in the base class.
    _chunkSize = config.getOption("data", "chunkSize").toInt();
}

// Creates a suitable device ready for reading.
QIODevice* ChunkerExample::newDevice()
{
    // Return an opened QUdpSocket.
    QUdpSocket* socket = new QUdpSocket;
    socket->bind(QHostAddress(host()), port());

    // Wait for the socket to bind.
    while (socket->state() != QUdpSocket::BoundState) {}
    return socket;
}

// Called whenever there is data available on the device.
void ChunkerExample::next(QIODevice* device)
{
    // Get a pointer to the UDP socket.
    QUdpSocket* udpSocket = static_cast<QUdpSocket*>(device);
    _bytesRead = 0;

    // Get writable buffer space for the chunk.
    WritableData writableData = getDataStorage(_chunkSize);
    if (writableData.isValid()) {
        // Get pointer to start of writable memory.
        char* ptr = (char*) (writableData.ptr());

        // Read datagrams for chunk from the UDP socket.
        while (isActive() && _bytesRead < _chunkSize) {
            // Read the datagram, but avoid using pendingDatagramSize().
            bool ok = udpSocket->hasPendingDatagrams();
```

```
            if (!ok) {
                // MUST WAIT for the next datagram.
                udpSocket->waitForReadyRead(100);
                continue;
            }
            qint64 maxlen = _chunkSize - _bytesRead;
            qint64 length = udpSocket->readDatagram(ptr + _bytesRead, maxlen);
            if (length > 0)
                _bytesRead += length;
        }
    }

    // Must discard the datagram if there is no available space.
    else {
        udpSocket->readDatagram(0, 0);
    }
}
```

4.1.6 Testing your Chunker

The *pelicanTest* library provides the *ChunkerTester* convenience class for easy unit/integration testing of your Chunker. See the API documentation for more details.

4.2 Data Clients

4.2.1 Introduction

Data clients retrieve and de-serialise (by use of adapters) binary data chunks for use with processing pipelines. Data clients are written by inheriting `AbstractDataClient` and implementing the `AbstractDataClient::getData()` method which much handle the data requirements of registered pipelines. The data client should therefore retrieve one or more binary data chunks, passing these to adapters for conversion to C++ container classes (called data blobs), and construct a set of adapted data suitable for running pipelines.

4.2.2 Overview

To create a new data client:

- Inherit the `AbstractDataClient` interface class.
- In the derived data client class, implement the `getData()` method. This method must retrieve a hash of data blobs suitable for running one of the registered pipelines. Data is typically retrieved as binary chunks of data and then de-serialised by Pelican data adapters.

4.2.3 Configuration

Data clients are supplied with a configuration node in their constructors. The configuration node resides in the `clients` section of the XML configuration file, and has a tag name which matches the class name of the data client.

Data clients all share some common configuration which is handled automatically by the data client factory. This takes the form of a set of tags which describe the mapping between the various data types handled by the client and their associated adapters.

For example:

```
<data type="VisibilityData" adapter="AdapterVisibilities"/>
<data type="AntennaPositions" adapter="AdapterAntennasPositions"/>
```

4.2.4 Specialised data clients

A number of specialised data clients, described in the following sections, are provided in the Pelican framework. These cover a number of common use cases and can either be used directly in your application or serve a starting point for a more specialised data client.

4.2.4.1 The PelicanServerClient class

The `PelicanServerClient` is an implementation of a data client for interfacing with the Pelican Server. Communication is made by TCP and makes use of the `PelicanClientProtocol` for communication with the server.

4.2.4.2 The FileDataClient class

The `FileDataClient` provides a specialist data client for accessing data directly from files rather than using the Pelican server. The list of files is specified in the XML configuration node for the client in the form of a list of tags called `data` with attributes `type` and `file` indicating the data type name (used to determine if the data categorised as service or stream data), and the file name respectively.

4.2.4.3 The DirectStreamDataClient class

The `DirectStreamDataClient` is a specialised data client that connects directly to one or more data streams without the need for a Pelican server, as described in the framework introduction. This client therefore provides a specialist mode of operation where handling of the input data streams is carried out in the same binary as processing pipelines. While this may impose limits on scalability, it has advantages where a high input data rate is handled continuously by a single pipeline.

4.3 Data Adapters

4.3.1 Introduction

Adapters provide a mechanism to convert chunks of raw binary data into the data members of a Pelican data blob (a specialised C++ container class for holding data used by the Pelican pipeline).

The most basic functionality of an adapter is to de-serialise chunks of data, although reordering and re-factoring of the data to a form that is convenient for subsequent pipeline processing may also be carried out.

Pelican currently provides support for two categories of adapters, distinguished by the type of input data chunks they are expected to process. These are stream adapters and service adapters (*## TODO: ref. description of stream and service data...*):

- **Stream Adapters:** De-serialise data chunks classed as stream data. Stream adapters allow the de-serialisation method to be modified by service data registered to be associated to the stream data.
- **Service Adapters:** De-serialise data chunks classed as service data. The adaption of data chunks classed as service data is not expected to change during a run of Pelican.

Adapters plug into Pelican data clients, and the de-serialise method is called for each iteration of the pipeline that requires data of the type they adapt. In addition to calling the adapter `deserialise()` method, the `config()` method of the adapter base class is called to set the pointer to the data blob which the adapter is to fill, the chunk size to be adapted, and pointers to any associated service data (in the case of stream adapters). This data is available to the `deserialise()` method through members of the base class, rather than explicitly though the interface of the `deserialise()` method.

4.3.2 Overview

To create a new adapter :

- Inherit from either the `AbstractStreamAdapter`, or the `AbstractServiceAdapter` class.
- In the derived class, implement the `deserialise()` method. This method is passed a pointer to an open `QIODevice` containing the serial chunk to be adapted. De-serialised data should be placed in the data blob pointer, `_data`, which is a protected member of the `AbstractAdapter` base class and set automatically by the data client on request for remote data in a pipeline to which adapter is providing de-serialised data blobs.
- The size of the chunk of data in the QIODevice (in bytes) as well as (in the case of stream adapters) associated service data blobs are available to the adapter by though data members of the base class, set by the data client _chunkSize and _serviceData.
- Adapters must register their existence with the adapter factory. Use the `PELICAN_DECLARE_ADAPTER()` macro under the class definition in the adapter's header file to register the adapter, supplying the name of the adapter class as the macro argument. Do not use quotes around the name.
- The data object being populated by the adapter must be a Pelican data blob (i.e. a class that inherits from `DataBlob`). There are a number of data blob types that are currently provided by the Pelican framework, so you may not need to create your own if these are suitable.

4.3.3 Configuration

All adapters must be supplied with a configuration node in their constructors. The configuration node resides in the adapters section of the configuration XML file, and has a tag name the same as the adapter class name.

The configuration within the node is then left up to the details of the specific adapter, but will need to contain all the information needed to de-serialise the chunk.

4.3.4 Example

The following example shows a stream adapter to convert chunks of data containing sections of a stream of real floating point values into the example data blob.

The XML configuration node is:

```
<AdapterExample>
    <samples number="512" bitsPerSample="8"/>
</AdapterExample>
```

The class definition is:

```
#ifndef ADAPTER_EXAMPLE_H
#define ADAPTER_EXAMPLE_H

#include "core/AbstractStreamAdapter.h"

using namespace pelican;

class AdapterExample : public AbstractStreamAdapter
{
    public:
        // Constructs the example adapter.
        AdapterExample(const ConfigNode& config);

        // Method to deserialise chunks of memory provided by the I/O device.
        void deserialise(QIODevice* in);

    private:
        unsigned _nBitsPerSample;
};

// Register the adapter.
PELICAN_DECLARE_ADAPTER(AdapterExample)

#endif // ADAPTER_EXAMPLE_H
```

and the class implementation is:

```
#include "reference/AdapterExample.h"
#include "reference/DataBlobExample.h"

// Construct the example adapter.
AdapterExample::AdapterExample(const ConfigNode& config)
    : AbstractStreamAdapter(config)
{
    // Read the configuration using configuration node utility methods.
    _nBitsPerSample = config.getOption("samples", "bitsPerSample").toUInt();
}

// Called to de-serialise a chunk of data in the I/O device.
void AdapterExample::deserialise(QIODevice* in)
{
    // A pointer to the data blob to fill should be obtained by calling the
    // dataBlob() inherited method. This returns a pointer to an
    // abstract DataBlob, which should be cast to the appropriate type.
    DataBlobExample* blob = (DataBlobExample*) dataBlob();

    // Set the size of the data blob to fill.
    // The chunk size is obtained by calling the chunkSize() inherited method.
    size_t nBytes = chunkSize();
    unsigned length = nBytes / _nBitsPerSample;
    blob->resize(length);

    // Create a temporary buffer for storing the chunk.
    std::vector<char> byteArray(nBytes);
```

```
    // Read the chunk from the I/O device.
    in->read(&byteArray[0], nBytes);

    // Get the pointer to the data array in the data blob being filled.
    float* data = blob->data();

    // Fill the data blob.
    for (unsigned i = 0; i < length; ++i) {
        data[i] = *reinterpret_cast<float*>(&byteArray[i * sizeof(float)]);
    }
}
```

4.4 Data Blobs

4.4.1 Introduction

DataBlobs are simply C++ structures that hold data for use by Pelican pipeline modules. They may contain arrays, blocks of memory and/or other data, and should provide methods to interact with that data. Their main function is to act as an interface between pipeline modules. Within a module, data is handled internally by whatever means necessary to carry out efficient computation, but modules should present a consistent interface and encapsulate their data requirements and outputs using a small number of data blobs.

DataBlobs cannot be pre-configured; i.e. they take no XML configuration data from the configuration file. Methods on the blob itself should be used to change its contents as required.

4.4.2 Overview

To create a new type of DataBlob:

- Inherit the DataBlob base class.
- There are no abstract methods that define the data storage, so you have considerable freedom in choosing the internal structure of the blob to suit the needs of the module(s) that will use it. Declare the required data structures in the blob's `private` section, and implement any `public` methods that will be used to interact with the data.
- If the blob needs to be sent to an output device, reimplement the `serialise()` method to send the contents of the blob to the device.
- If the blob needs to be filled from an input device, reimplement the `deserialise()` method to read the contents of the blob from the device.
- Data blobs must register their existence with the blob factory. Use the `PELICAN_DECLARE_DATABLOB()` macro under the class definition in the data blob's header file to register it, supplying the name of the data blob class as the macro argument. Do not use quotes around the name.

4.4.3 Example

This example creates and declares a new data blob of floating-point data.

```
#ifndef DATABLOBEXAMPLE_H
#define DATABLOBEXAMPLE_H

#include "data/DataBlob.h"
#include <vector>

using namespace pelican;

/*
 * This data blob holds an array of floating-point data.
 */
class DataBlobExample : public DataBlob
{
    public:
        // Constructs an example data blob.
        DataBlobExample() : DataBlob("DataBlobExample") {}

        const float* data() const { return (_data.size() > 0 ? &_data[0] : 0);
      }

        float* data() { return (_data.size() > 0 ? &_data[0] : 0); }

        // Resizes the data store provided by the data blob.
        void resize(unsigned length) { _data.resize(length); }

        // Returns the size of the data.
        unsigned size() const { return _data.size(); }

    private:
```

```
        std::vector<float> _data; // The actual data array.
};

PELICAN_DECLARE_DATABLOB(DataBlobExample)

// Other example data blobs used by the example pipeline.
typedef DataBlobExample DataBlobExample1;
typedef DataBlobExample DataBlobExample2;

PELICAN_DECLARE_DATABLOB(DataBlobExample1)
PELICAN_DECLARE_DATABLOB(DataBlobExample2)

#endif // DATABLOBEXAMPLE_H
```

4.5 Pipeline Modules

4.5.1 Introduction

Modules are the components of Pelican pipelines that perform intensive computational operations on the data flowing through them. Normally, several modules would be run (one after another) to build up a processing pipeline. A module may contain anything: it can be as simple or as complex as needed to perform the task required of it, and use (for example) multiple threads, OpenMP, CUDA or OpenCL (if available) to help speed up computation.

DataBlobs (Pelican data structures) are used to pass data between pipeline modules. Depending on your needs, you may have to design and implement a new type of data blob to work with a new module.

4.5.2 Overview

To create a new pipeline module:

- Inherit the `AbstractModule` class.
- There are no abstract methods on the module to perform computation, so you have considerable freedom to define the interfaces required. By convention, a `run()` method should be called with one or more input and output data blobs as function arguments.
- Modules must register their existence with the module factory. Use the PELICAN_DECLARE_MODULE() macro in the under the class definition in the module's header file to register the module, supplying the name of the module class as the macro argument. Do not use quotes around the name.

4.5.3 Configuration

The `AbstractModule` interface ensures that modules are passed a configuration node of the XML document when constructed, so that any initial parameters can be set up as required.

The configuration node resides in the `modules` section of the XML file, and has a tag name the same as the module class name.

Since the base class does not check for common options, there are currently no standard XML tags for modules. However, there are some convenience tags.

Frequency channels can be specified in the XML configuration as

```
<channels>1,5,182</channels>
```

or, for many contiguous channels,

```
<channels start="0" end="511"/>
```

To obtain this channel list, there is a convenience method ConfigNode::getUnsignedList(), which returns the list of channel indices in a standard vector.

```
std::vector<unsigned> = configNode.getUnsignedList("channels");
```

4.5.4 Example

This example creates a module to perform a trivial operation on two floating-point vectors. As input, it takes two example data blobs, and fills an output data blob according to the given configuration. The configuration specifies whether the input vectors should be added or multiplied together.

The XML contents of its configuration node are:

```
<ModuleExample>
    <operation type="add"/>
</ModuleExample>
```

The class definition is:

```
#ifndef MODULEEXAMPLE_H
#define MODULEEXAMPLE_H

#include "core/AbstractModule.h"

using namespace pelican;

class DataBlobExample;

/*
 * A simple example to demonstrate how to write a pipeline module.
 */
class ModuleExample : public AbstractModule
{
    public:
        ModuleExample(const ConfigNode& config);
        void run(const DataBlobExample* input1,
                const DataBlobExample* input2, DataBlobExample* output);

    private:
        typedef enum {OpNone, OpAdd, OpMultiply} Operation;
        Operation optype;
};

// Register the module.
PELICAN_DECLARE_MODULE(ModuleExample)

#endif // MODULEEXAMPLE_H
```

and the class implementation is:

```
#include "reference/ModuleExample.h"
#include "reference/DataBlobExample.h"
#include "utility/Config.h"

// Construct the example module.
ModuleExample::ModuleExample(const ConfigNode& config)
    : AbstractModule(config)
{
    // Initialise.
    optype = OpNone;

    // Set operation type from the configuration.
    QString type = config.getOption("operation", "type").toLower();
    if (type == "add")
        optype = OpAdd;
    else if (type == "multiply")
        optype = OpMultiply;
}

// Runs the module.
void ModuleExample::run(const DataBlobExample* input1,
        const DataBlobExample* input2, DataBlobExample* output)
{
    // Ensure both input data blobs match in dimensions.
    if (input1->size() != input2->size())
        throw QString("ModuleExample::run(): Input data dimension mismatch.");

    // Ensure the output storage data is big enough.
    unsigned nPts = input1->size();
    if (output->size() != nPts)
        output->resize(nPts);

    // Get pointers to the memory to use.
    const float* in1 = input1->data();
    const float* in2 = input2->data();
    float* out = output->data();

    // Perform the operation.
    if (optype == OpAdd) {
        for (unsigned i = 0; i < nPts; ++i) {
            out[i] = in1[i] + in2[i];
        }
    }
    else if (optype == OpMultiply) {
        for (unsigned i = 0; i < nPts; ++i) {
            out[i] = in1[i] * in2[i];
```

```
            }
        }
}
```

4.6 Pipelines

4.6.1 Introduction

Pelican pipelines act as containers for the data processing framework. A pipeline is simply a C++ class that defines the operations performed on a single chunk of stream data, and is run repeatedly by the pipeline driver whenever there is a new chunk of data available to be processed. The data processing itself actually happens within pipeline modules.

There are no default pipelines: in order to process data using Pelican, you will need to write your own pipeline class. Fortunately, this is very simple.

4.6.2 Overview

To create a new pipeline:

- Inherit from the `AbstractPipeline` class, which defines the pipeline interface.
- In the derived class, implement the `init()` method to perform one-off initialisation, such as creating pipeline modules, and to request the remote data required to run the pipeline.
- In the derived class, implement the `run()` method to define the operations carried out for each iteration of the pipeline.
- There are some protected methods in `AbstractPipeline` (detailed below) that should be used when setting up the pipeline.

4.6.2.1 The init() method

Here you must specify the types of data required using the `requestRemoteData()` protected method. It takes at least one argument, which is a string containing the type-name (the class name) of the required data blob. The method should be called as many times as required to fulfil all remote data requirements. An optional second argument allows you to specify the number of DataBlobs of this type to store in a history buffer. Using a history buffer allows your pipeline to refer to previous DataBlobs (via the () method) without incuring the performance penalty of an unnessasary data copy.

One-off initialisation, such as creating pipeline modules, should also be performed here. Use the `createModule()` protected method with the module type (class name) and an optional given name (if using a configuration file containing multiple different configurations for the same module type). The method returns an `AbstractModule` pointer, which should be stored and used to call methods on the module. Remember to call delete on this pointer in the destructor.

If there are any data blobs that exist local to the pipeline (i.e. that are not provided as part of the remote data), then they should also be created here using the `createBlob()` protected method. The method returns a `DataBlob` base class pointer. You can use the <type> method to create a `QList` of datablobs all at once.

Configuration information, if required, can be specified inside the <pipelineConfig> tag of the pipelines xml file. The appropriate `ConfigNode` object can be accessed through a call to the () protected method.

4.6.2.2 The run() method

This is where data processing happens, and is called each time a new data chunk is available. The method is supplied with a hash of data blob pointers to the remote data requested in the `init()` method: this hash of pointers is updated each time the method is called. The hash keys are of type `QString`, and correspond to the class names of the data blobs. To get the data blob pointers, look them up in the hash using their names with the [] dereference operator (see example, below).

The data blob pointers are then passed to the pipeline modules as required. Since the pipeline is just a C++ class, it can be used just like any other: private state variables can be used if required, and the pipeline can call methods on any objects known to it. The pipeline modules simply provide a convenient way to encapsulate functionality.

However, it is important to remember that the `run()` method should only be used to define a single iteration of the pipeline. The method must exit before the next chunk of data can be processed.

Pipelines must be registered with the pipeline driver in `main()`: see the section on writing main() for more details.

4.6.3 Example

In the following, a new pipeline is created to generate an image from remote visibility data. The `init()` method creates the pipeline modules using the `createModule()` protected method, creates the local image data blob using `createBlob()`, and requests remote data to be supplied to the pipeline using the `requestRemoteData()` method.

This example creates a new image each time the `run()` method is called. The modules are configured using their own settings from the XML configuration file.

The class definition is:

```
#ifndef PIPELINEEXAMPLE_H
#define PIPELINEEXAMPLE_H

#include "core/AbstractPipeline.h"

using namespace pelican;

class ModuleExample;
class DataBlobExample;

class PipelineExample : public AbstractPipeline
{
    public:
        PipelineExample();
        ~PipelineExample();
        virtual void init();
        virtual void run(QHash<QString, DataBlob*>& remoteData);

    private:
        // Module pointers.
        ModuleExample* adder;
        ModuleExample* multiplier;

        // Local data blob pointers.
        DataBlobExample* outputData;

};

#endif // PIPELINEEXAMPLE_H
```

and the class implementation is:

```
#include "reference/PipelineExample.h"
#include "reference/ModuleExample.h"
#include "reference/DataBlobExample.h"

PipelineExample::PipelineExample()
    : AbstractPipeline(), adder(0), multiplier(0), outputData(0)
{
}

// Destructor
// clean up and created Modules and DataBlobs:w
//
PipelineExample::~PipelineExample()
{
   delete adder;
   delete multiplier;
   delete outputData;
}

// Initialises the pipeline, creating any modules and data blobs required,
// and requesting any remote data.
void PipelineExample::init()
{
```

```
    // Create the pipeline modules.
    adder = (ModuleExample*) createModule("ModuleExample", "adder");
    multiplier = (ModuleExample*) createModule("ModuleExample", "multiplier");

    // Create local data blobs.
    outputData = (DataBlobExample*) createBlob("DataBlobExample");

    // Request remote data.
    requestRemoteData("DataBlobExample1");
    requestRemoteData("DataBlobExample2");
}


// Defines a single iteration of the pipeline.
// This performs the operation on the two vectors x and y such that
// the output is given by x * (x + y)
void PipelineExample::run(QHash<QString, DataBlob*>& remoteData)
{
    // Get pointers to the remote data blobs from the supplied hash.
    DataBlobExample1* x = (DataBlobExample1*) remoteData["DataBlobExample1"];
    DataBlobExample2* y = (DataBlobExample2*) remoteData["DataBlobExample2"];

    // Run each module as required.
    adder->run(x, y, outputData);
    multiplier->run(x, outputData, outputData);
}
```

4.7 Output Streamers

4.7.1 Overview

You will need to send your processed data elsewhere, perhaps to a storage system, a monitoring and control system or a further processing pipeline. Pelicans Output Manager allows you to separate the details of the destination of the data from the specification of what has to be written out.

The OutputManager uses a plug-in mechanism to convert DataBlobs into the formats required. These plug-ins are called OutputStreamers and must implement the *AbstractOutputStream* interface.

4.7.1.1 The Pelican DataBlobFile format

The DataBlobFile streamer writes any DataBlob that supports the serialise() and deserialise() methods to a binary format file. Classes are provided to read this file format and convert it back into DataBlobs (e.g. the *DataBlobFile-Reader*). Two basic file formats are recognised: homogeneous, and heterogeneous. The *homogeneous* file format stores DataBlobs of a single type, whereas the heterogeneous format allows the storage of more than one DataBlob type in the same file at a cost of extra storage space.

4.7.1.2 The TCP Blob Server (PelicanTCPBlobServer)

This streamer sets up a TCP/IP streaming server to supply any connecting client with a copy of the data as it becomes available (server push model). You can use this, in connection with the *DataBlobClient* class, to move data to another machine or process for storage, further processing, online monitoring, etc.

4.7.2 Custom OutputStreamers

As already mentioned, the *AbstractOutputStreamer* provides the base class for plug-ins into the OutputManager. To provide new output functionality, you simply inherit from this and implement the virtual *sendStream()* method.

We provide an example that takes the *DataBlobExample* object and converts it to an entry in a comma seperated value(csv) file.

```
#ifndef OUTPUTSTREAMEXAMPLE_H
#define OUTPUTSTREAMEXAMPLE_H

#include <QtCore/QList>

class QIODevice;

#include "output/AbstractOutputStream.h"

using namespace pelican;

class OutputStreamExample : public AbstractOutputStream
{
    public:
        // OutputStreamExample constructor.
        OutputStreamExample(const ConfigNode& configNode);

        // OutputStreamExample destructor.
        ~OutputStreamExample();

        // Add a file for output to be saved.
        void addFile(const QString& filename);

    protected:
        // Sends the data blob to the output stream.
        void sendStream(const QString& streamName, const DataBlob* dataBlob);

    private:
        QList<QIODevice*> _devices;
};

PELICAN_DECLARE(AbstractOutputStream, OutputStreamExample)

#endif // OUTPUTSTREAMEXAMPLE_H
```

```
#include "tutorial/OutputStreamExample.h"

#include <QtCore/QFile>
#include <QtCore/QIODevice>
#include <QtCore/QTextStream>
#include <iostream>

#include "tutorial/SignalData.h"
#include "utility/ConfigNode.h"

// Constructs the output stream.
OutputStreamExample::OutputStreamExample(const ConfigNode& configNode)
    : AbstractOutputStream(configNode)
{
    // Get the filename from the configuration node, and open it for output.
    QList<QString> fileNames = configNode.getOptionList( "file", "name");
    foreach( const QString& filename, fileNames ) {
        addFile(filename);
    }
}

// Destroys the output stream, deleting all the devices it uses.
OutputStreamExample::~OutputStreamExample()
{
    foreach (QIODevice* device, _devices) {
        delete device;
    }
}

// Adds a file to the output stream and opens it for writing.
void OutputStreamExample::addFile(const QString& filename)
{
    verbose(QString("Creating file %1").arg(filename));
    QFile* file = new QFile(filename);
    if (file->open(QIODevice::WriteOnly)) {
        _devices.append(file);
    }
    else {
        std::cerr << "Cannot open file for writing: "
                << filename.toStdString() << std::endl;
        delete file;
    }
}

// Sends the data blob to the output stream.
void OutputStreamExample::sendStream(const QString& /*streamName*/,
        const DataBlob* blob)
{
    // Check it's a data blob type we know how to use.
    if (blob->type() != "SignalData") return;

    const float* data = ((const SignalData*)blob)->ptr();
    unsigned size = ((const SignalData*)blob)->size();
    if (data)
    {
        // Construct the comma separated value string.
        QString csv = QString::number(data[0]);
        for (unsigned i = 1; i < size; ++i) {
            csv += "," + QString::number(data[i]);
        }

        // Output the string to each file
        foreach (QIODevice* device, _devices) {
            QTextStream out(device);
            out << csv << endl;
        }
    }
}
```

4.7.2.1

To use your new class, you must set up your OutputManager correctly. See the Data Output section of the "Getting Started Guide" for an example. Make sure that you #include the header for this class somewhere in order to register it with the OutputStream factory.

4.8 Configuration Files

4.8.1 Introduction

The configuration file allows control over all aspects of the runtime binary. As well as things such as server IP addresses and port information, the configuration file is also used to set parameters for individual modules in the pipeline.

There are currently no default locations for pelican to search for configuration files. You must specify the configuration file on the command line with the `-config` option. Note that this may change in future revisions.

4.8.2 File Structure

The configuration files are XML-based, and can be edited by hand using any text editor. The document type is `pelican` and the root node tag is `configuration`, which has a single `version` attribute. The current configuration version is 1.0.

The most basic valid configuration file is:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE pelican>
<configuration version="1.0">
</configuration>
```

Inside the `configuration` tag there can be up to three sections, delimited by their own tags:

- `pipeline`
- `server`
- `nodesets`

Some or all of these sections may be present. The *pipeline* section contains configuration details for the pipeline binary. The *server* section contains configuration details for the server binary. *Nodesets* define groups of XML nodes that can be copied into specific locations using an `import` tag.

There are a number of sub-sections allowed within the *pipeline* and *server* tags.

Under *pipeline*, any of the following can be present in any order:

- `clients` (configuration for data clients)
- `chunkers` (configuration for data chunkers, if using a direct stream client)
- `adapters` (configuration for data adapters)
- `modules` (configuration for pipeline modules)

Under *server*, any of the following can be present in any order:

- `chunkers` (configuration for data chunkers)
- `buffers` (configuration for data buffers)

Within these sub-sections, the tag names must match the names of the objects (i.e. the class names). Multiple different objects of the same type that need different configurations can be distinguished by using the `name` attribute in the class tag.

4.8.2.1 Example

This is an example configuration file that contains both *pipeline* and *server* sections. There are two chunkers of the same type that must connect to different UDP ports, so the `name` attribute is used to disambiguate them.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE pelican>
<configuration version="1.0">
<pipeline>
    <clients>
        <PelicanServerClient>
            <server host="127.0.0.1" port="2000"/>
            <data type="VisibilityData" adapter="AdapterStationVisibilities"/>
       </PelicanServerClient>
    </clients>

    <adapters>
        <AdapterStationVisibilities>
            <antennas number="2"/>
            <channels start="0" end="1"/>
            <polarisation value="both"/>
            <dataBytes number="8"/>
        </AdapterStationVisibilities>
    </adapters>
</pipeline>

<server>
    <buffers>
       <VisibilityData>
           <buffer maxSize="10000" maxChunkSize="10000"/>
       </VisibilityData>
    </buffers>

    <chunkers>
       <TestUdpChunker name="1">
           <connection host="127.0.0.1" port="2002"/>
           <data type="VisibilityData" chunkSize="512"/>
       </TestUdpChunker>
       <TestUdpChunker name="2">
           <connection host="127.0.0.1" port="2003"/>
           <data type="VisibilityData" chunkSize="512"/>
       </TestUdpChunker>
    </chunkers>
</server>
</configuration>
```

4.8.3 Importing Configuration Data

The only reserved global tag name is the `import` tag, which can be used to make preprocessor-like node substitutions. This can be useful if, for example, modules share common configuration options.

Import tags may be used anywhere under the root node, so the imported nodes will be appended as children of the tag's parent node. The import source may be a special set of nodes (a nodeset) or another XML configuration file.

4.8.3.1 Importing Nodesets

Groups of common nodes can be declared as nodesets by enclosing them inside `nodeset` tags within the `nodesets` section. Each nodeset must be given a unique name using the `name` attribute, so the nodeset can be copied into other locations in the document using an `import nodeset="[name]"` tag. Nodes within the nodeset will be appended to the end of the list of child nodes of the enclosing parent regardless of where they appear, so that local settings will override those from the nodeset if there is a conflict. It is possible for nodesets themselves to import nodes from other nodesets.

4.8.3.1.1 Example

For brevity, the XML header and root node have been omitted from this example.

```
<pipeline>
    <modules>
        <Module1>
            <parameter value="1"/>
            <import nodeset="set1"/>
        </Module1>
        <Module2>
            <import nodeset="set1"/>
            <parameter value="2"/>
        </Module2>
    </modules>
</pipeline>
<nodesets>
    <nodeset name="set1">
        <common1 value="first common parameter"/>
        <common2 value="second common parameter"/>
    </nodeset>
</nodesets>
```

The preprocessor will transform this into the following XML:

```
<pipeline>
    <modules>
        <Module1>
            <parameter value="1"/>
            <!--% Imported nodeset set1-->
            <common1 value="first common parameter"/>
            <common2 value="second common parameter"/>
        </Module1>
        <Module2>
            <parameter value="2"/>
            <!--% Imported nodeset set1-->
            <common1 value="first common parameter"/>
            <common2 value="second common parameter"/>
        </Module2>
    </modules>
</pipeline>
<nodesets>
    <nodeset name="set1">
        <common1 value="first common parameter"/>
        <common2 value="second common parameter"/>
    </nodeset>
</nodesets>
```

4.8.3.2 Importing Files

It is possible to import all the child nodes under the root node of a document in another file.

To do this, use the `import file="[file name]"` tag. If not an absolute path, the file name must be relative to the current working directory.

4.8.3.3 Importing Remote Files

The `import url="[address]"` tag is not currently implemented, but is reserved for future use.

4.8.4 Obtaining Configuration Data

Objects that are configurable (which are currently pipeline modules, adapters, chunkers and data clients) can obtain their settings using convenience methods on the ConfigNode object passed to them in their constructors.

To get the text for a single option, call ConfigNode::getOption() with the tag name and attribute name. The method returns a QString containing the required data.

For example, a module might have this configuration XML in the `modules` section:

```
<Module1>
    <option value="option"/>
    <parameter value="1.0"/>
</Module1>
```

The configuration node for Module1 will be passed down in the module's constructor. To get the parameter value, one could use:

```
double param = configNode.getOption("parameter", "value").toDouble();
```

4.8.5 Common Module Options

In radio astronomy, it is usual to have to specify a subset of radio frequencies and/or channel indices when working with radio telescope data. This channel information often has to be supplied to many different processing modules.

Frequency channels are specified in the XML configuration as

```
<channels>1,5,182</channels>
```

or, for many contiguous channels,

```
<channels start="0" end="511"/>
```

To obtain this channel list, there is a convenience method ConfigNode::getUnsignedList(), which returns the list of channel indices in a standard vector.

```
std::vector<unsigned> = configNode.getUnsignedList("channels");
```

4.8.6 Common Chunker Options

Chunkers can use any device, but usually work with network sockets. For this reason, the chunker base class will look for a `connection` tag in its configuration node, with attributes `host` (the hostname or IP address) and `port` number. This XML tag should be provided for all chunkers that use network sockets.

```
<connection host="127.0.0.1" port="2001"/>
```

All chunkers must also declare the data type that they provide using the `data type="[type]"` tag, for example:

```
<data type="VisibilityData"/>
```

The data type string is the name of the data blob that will be eventually filled by the adapter.

4.8.7 Common Data Client Options

Data clients that connect to the Pelican server must specify the hostname and port for the TCP connection, using the `server` tag:

```
<server host="127.0.0.1" port="2000"/>
```

For each data type that the client can handle, there must also be a corresponding adapter to deserialise the data stream into data blobs. Use a `data` tag with the attributes `type` and `adapter` so that the client knows which adapter to use for each data type.

```
<data type="VisibilityData" adapter="MyAdapter"/>
```

When you have multiple adapters of the same type you can use the `name` variable to specify a specific adapter configuration.

```
<data type="VisibilityData" adapter="MyAdapter" name="configuration1"/>
```

There should be as many `data` tags as necessary, one for each data type.

4.9 Data Emulation

4.9.1 Introduction

Synthetic packets of data are often required when testing new data chunkers. These synthetic data packets can be generated conveniently using the classes provided in the data emulation framework.

The EmulatorDriver class takes care of the common, low-level details. It creates a separate thread in which to run the data emulator, and is responsible for writing the data packets to the output device. On construction, the EmulatorDriver takes ownership of a pointer to an AbstractEmulator derived class, which is called repeatedly by the driver to generate the required data packets. The emulator itself will be destroyed with the EmulatorDriver object, so it must not be explicitly deleted.

The primary function of the AbstractEmulator is to provide a pointer to a block of memory to use when sending data packets.

4.9.2 Overview

To create a new data emulator:

- Create a derived class of a suitable abstract emulator. If a UDP packet emulator is required, then inherit `AbstractUdpEmulator`; if not, then inherit `AbstractEmulator`.
- In the derived class, ensure that a pointer to an open input device is returned from the `createDevice()` method. The `AbstractUdpEmulator` already takes care of this, but this method must be implemented for `AbstractEmulator` derived classes. The base class takes ownership of the device, so it must not be deleted explicitly.
- In the derived class, implement the `getPacketData()` method. The method must update the input parameters to set a pointer to a block of memory to use for the packet, and the packet size in bytes. This method fills the packet with emulated data.

There are a number of other options that may be set via virtual methods in the derived class:

- To make the data emulator start sending data automatically on construction of the driver, reimplement the `autostart()` method to return `true` or `false` (default true). If false, then call start() on the EmulatorDriver to start the emulator.
- To set the duration between calls to getPacketData(), implement the `interval()` method to return the interval in microseconds between packets (default 100000).
- To set the number of packets sent by the emulator, reimplement the `nPackets()` method and return the number of packets required. If the number of packets is negative, then the emulator will run forever (default -1).
- To make the emulator wait before sending packets, reimplement the `startDelay()` method to return the number of seconds to wait initially (default 0).

4.9.3 Configuration

The emulator can take a configuration node in its constructor if required. The method `ConfigNode::setFromString()` can be used to set the XML contents of the configuration node.

Classes derived from `AbstractUdpEmulator` must supply a configuration node containing at least the host address and port number. Use the `connection` tag with `host` and `port` attributes, as in the following example.

4.9.4 Example

In the following, a new emulator is defined to send packets of real-valued UDP data down a socket. The XML contents of its configuration node are:

```
<EmulatorExample>
    <connection host="127.0.0.1" port="2002"/>
    <packet size="512" interval="1000" initialValue="0.1"/>
</EmulatorExample>
```

The class definition is:

```
#ifndef EMULATOREXAMPLE_H
#define EMULATOREXAMPLE_H

#include "emulator/AbstractUdpEmulator.h"
#include <QtCore/QByteArray>

namespace pelican {

class ConfigNode;

/*
 * This emulator outputs simple packets of real-valued double-precision data
 * using a UDP socket.
 * It should be constructed with a configuration node that contains
 * the packet size in bytes, the interval between packets in microseconds,
 * and the initial data value.
 *
 * The default values are:
 *
 * <packet size="8192" interval="100000" initialValue="0" />
 */
class EmulatorExample : public AbstractUdpEmulator
{
    public:
        // Constructs a new UDP packet emulator.
        EmulatorExample(const ConfigNode& configNode);

        // Destroys the UDP packet emulator.
        ~EmulatorExample() {}

        // Creates a UDP packet.
        void getPacketData(char*& ptr, unsigned long& size);

        // Returns the interval between packets in microseconds.
        unsigned long interval() {return _interval;}

    private:
        double _initialValue;      // The initial value of the packet data.
        unsigned long _packetSize; // The size of the packet.
        unsigned long _interval;   // The interval between packets in
       microseconds.
        QByteArray _packet;        // The data packet.
        unsigned long _counter;    // Packet counter.
};

} // namespace pelican

#endif // EMULATOREXAMPLE_H
```

and the class implementation is:

```
#include "reference/EmulatorExample.h"
#include "utility/ConfigNode.h"

namespace pelican {

EmulatorExample::EmulatorExample(const ConfigNode& configNode)
    : AbstractUdpEmulator(configNode)
{
    // Initialise members.
    _counter = 0;
    _interval = configNode.getOption("packet", "interval", "100000").toULong();
    _packetSize = configNode.getOption("packet", "size", "8192").toULong();
    _initialValue = configNode.getOption("packet", "initialValue", "0").
      toDouble();

    // Check that the packet size is a multiple of 8.
    if (_packetSize % sizeof(double) != 0)
```

```
        throw QString("EmulatorExample: "
                "Packet size must be a whole number of doubles.");

    // Check that the packet size is not empty.
    if (_packetSize == 0)
        throw QString("EmulatorExample: Packet size must be > 0.");

    // Initialise the packet.
    _packet.resize(_packetSize);
}

void EmulatorExample::getPacketData(char*& ptr, unsigned long& size)
{
    // Set the output data.
    ptr = _packet.data();
    size = _packetSize;

    // Fill the packet.
    unsigned nDoubles = _packet.size() / sizeof(double);
    for (unsigned i = 0; i < nDoubles; ++i)
        reinterpret_cast<double*>(ptr)[i] = _initialValue + _counter;

    // Increment counter for next time.
    _counter++;
}

} // namespace pelican
```

To use the emulator, invoke the emulator driver as follows:

```
ConfigNode emulatorConfig(xmlString);
EmulatorDriver emulator(new EmulatorExample(emulatorConfig));
```

4.10 Building Pelican Binaries

4.10.1 Introduction

This section of the documentation describes how to write binaries making use of the Pelican framework. Pelican naturally splits into two distinct categories of binaries; those handling and serving input data streams (called server binaries), and those responsible for de-serialising data and executing processing pipelines (called pipeline binaries). The following sections describe how write simple binaries for these two categories.

4.10.2 Pipeline Binaries

The pipeline binary contains the data client, the pipeline driver, and one or more pipelines. Like any other C++ program, execution begins in `main()`.

Since Pelican makes extensive use of the excellent cross-platform `Qt Library` for its data structures, timers and network modules, a `QCoreApplication` object must be created before anything else. Almost all of the Pelican functionality is encapsulated by the `PipelineApplication` class. Both of these application objects require the command line arguments to be passed to their constructors.

4.10.2.1 Single Pipeline

For each pipeline that must be run, call the `PipelineApplication::registerPipeline()` method on the pipeline application object. The method requires a pointer to an `AbstractPipeline` object (the base class of all Pelican pipelines) as its argument.

```
#include "core/PipelineApplication.h"
#include "tutorial/SignalProcessingPipeline.h"
#include <QtCore/QCoreApplication>

// Include any headers that are referenced by name.
#include "tutorial/OutputStreamExample.h"
#include "tutorial/SignalDataAdapter.h"

int main(int argc, char* argv[])
{
    // Create a QCoreApplication.
    QCoreApplication app(argc, argv);

    try {
        // Create a PipelineApplication.
        PipelineApplication pApp(argc, argv);

        // Register the pipelines that can run.
        pApp.registerPipeline(new SignalProcessingPipeline);

        // Set the data client.
        pApp.setDataClient("PelicanServerClient");

        // Start the pipeline driver.
        pApp.start();
    }

    // Catch any error messages from Pelican.
    catch (const QString& err) {
        std::cerr << "Error: " << err.toStdString() << std::endl;
    }

    return 0;
}
```

4.10.2.2 Multiple Pipelines

It is possible to configure more than one pipeline to be operational in a single Pelican pipeline binary. The appropriate pipeline will be run, depending on the data that is available.

Multiple pipelines that require the same remote data sets are, however, *not* supported for efficiency reasons, as copies of potentially very large data sets would need to be made for each pipeline.

4.10.3 Server Binary

Pelican server binaries are responsible for receiving, splitting and serving one or more incoming data streams. As such, server binaries are constructed from a Pelican server (`PelicanServer`), one or more Data Chunkers and one or more protocols for handling communications between pipelines and the server (e.g. `PelicanProtocol`).

The following code illustrates how to create a simple server binary for use with the example code presented in the getting started tutorial.

```
#include "server/PelicanServer.h"
#include "comms/PelicanProtocol.h"
#include "utility/Config.h"

#include "tutorial/SignalChunker.h"

#include <QtCore/QCoreApplication>
#include <iostream>

using namespace pelican;

int main(int argc, char ** argv)
{
    // 1. Create a QCoreApplication.
    QCoreApplication app(argc, argv);

    // 2. Create a Pelican configuration object (this assumes that a Pelican
    // configuration XML file is supplied as the first command line argument)
    if (argc != 2) {
        std::cerr << "Please supply an XML config file." << std::endl;
        return 0;
    }
    QString configFile(argv[1]);
    Config config(configFile);

    try {
        // 3. Create a Pelican server.
        PelicanServer server(&config);

        // 4. Attach the chunker to server.
        server.addStreamChunker("SignalChunker");

        // 5. Create a communication protocol object and attach it to the
       server
        // on port 2000.
        AbstractProtocol *protocol =  new PelicanProtocol;
        server.addProtocol(protocol, 2000);

        // Start the server.
        server.start();

        // 6. When the server is ready enter the QCoreApplication event loop.
        while (!server.isReady()) {}
        return app.exec();
    }

    // 7. Catch any error messages from Pelican.
    catch (const QString& err)
    {
        std::cerr << "Error: " << err.toStdString() << std::endl;
        return 1;
    }
}
```

4.10.3.1 Notes (refer to references in the code):

1. Pelican makes extensive use of the `Qt Library` and as such requires a QtCoreApplication or QApplication for its main event loop.
2. Pelican configuration (XML) objects can be constructed by loading a configuration file or programmatically.
3. Create a Pelican Server, the main data server class used to spawn sessions which handle incoming data requests.
4. Attach chunkers to process the incoming stream(s) into discrete partitions. Chunkers are specified by their class name and constructed automatically by a factory.
5. Attach a protocol for communication with a Pelican server client, the PelicanProtocol is a basic low overhead protocol which can be usef for this.

6. It is necessary to wait for the server thread to have entered its event loop before starting the application main event loop as the start method on the server returns immediately.
7. Pelican classes throw helpful QString messages on various errors so it is often useful to catch these.

4.10.4 The direct stream client

As described in the introduction to the Pelican framework. Pelican can also be run in a direct input mode using the a data client derived from the `DirectStreamDataClient` which embeds chunkers directly into the data client framework. In this mode binaries are written in the same was as the pipeline binary example.

4.11 Building the Pelican library

Pelican uses a CMake (www.cmake.org) build system and requires CMake 2.4 or higher.

In order to build the Pelican library (example commands are given for Linux systems):

1. Create a folder for the build.

   ```
   mkdir build
   ```

2. Move into the build folder.

   ```
   cd build
   ```

3. Run CMake and point at the top level CMakeLists.txt file.

   ```
   cmake ../path/to/CMakeLists.txt [BUILD OPTIONS (see below)]
   ```

4. Run make to build the code.

   ```
   make
   ```

5. [optional] Install Pelican (may need root permission depending on the specified install location).

   ```
   make install
   ```

In order to check if the Pelican build was successful run `Make test` from the top level build directory to launch the set of unit tests.

4.11.1 Build Options

When running cmake to construct the makefiles there are a number of build options that can be selected.

- **-DCMAKE_BUILD_TYPE =** release or debug(default)
  Build in release or debug mode (changes compiler flags).
- **-DCMAKE_INSTALL_PREFIX =** path
  Root directory used to install files when `make install`.
- **-DLIBRARY_INSTALL_DIR =** path (default=lib)
  Library install location appended to the install prefix.
- **-DINCLUDE_INSTALL_DIR =** path (default include/pelican)
  Header file install location appended to the install prefix.
- **-DCMAKE_CXX_COMPILER =** complier (default: c++)
  Sets the C++ compiler.
- **-DCMAKE_C_COMPILER =** compiler (default: gcc)
  Sets the C compiler.
- **-DTIMER =** true or false(default)
  Enable additional debug timer.
- **-DBUILD_STATIC =** off(default) or on
  Build static versions of libraries.

4.12 Testing Utilities

In addition to the core framework, Pelican includes a utility library (`pelican-testutils`) consisting of a number of classes and functions which are designed to be used in testing code written using Pelican.

The test utility classes derive from the extensive unit testing central to the development of Pelican, and examples of there use can be found in the unit tests in the source code distribution of the library as well as in relevant sections of the reference documentation.

4.12.1 The Pelican test utility library

The test utility library currently provides a number of classes listed below.

- Utility
  - TestConfig
  - SocketTester
- Data
  - TestDataBlob
- Core
  - TestPipeline
  - TestDataClient
  - TestServiceAdapter
  - TestStreamAdapter
- Server
  - TestChunker
  - TestUdpChunker
  - TestProtocol
  - TestServer
  - PelicanTestClient
  - ChunkerTester
- Emulator
  - RealUdpEmulator
- Output
  - TestOutputStreamer:
  - TestDataBlobClient
- Viewer
  - TestDataViewerWidget

4.12.2 Testing with CppUnit

Unit tests used in the development of Pelican make use of the the CppUnit test framework (documentation can be found at `http://cppunit.sourceforge.net/doc/lastest/index.html`) which provides a simple framework for developing and running tests either individually or as part of a test suite.